MADURAI KAMARAJ UNIVERSITY

(University with Potential for Excellence)

DISTANCE EDUCATION

B.Sc., (Mathematics)

THIRD YEAR

ANCILLARY PAPER - II

PROGRAMMING IN C AND C++

Recognized By DEC

www.mkudde.org

S 92

MADURAI KAMARAJ UNIVERSITY

(University with Potential for Excellence)

DISTANCE EDUCATION

B.Sc. (MATHEMATICS)

Third Year

ANCILLARY PAPER - II

PROGRAMMING IN C AND C++

Recognized by DEC

www.mkudde.org

S 92

JEGN/1000 COPIES

B.Sc (Mathematics)
Third Year

Ancillary Paper – II
PROGRAMMING IN C AND C++

Welcome

Dear Students,

We welcome you as a student of the Final year B.Sc degree course.

This paper deals with the subject 'PROGRAMMING IN C AND C++'.

The learning material for this paper will be supplemented by contact lectures.

In this book the first five units deal with C Programming and the last five units deal with C++ Programming.

Learning through the Distance Education mode, as you are all aware, involves self learning and self assessment and in this regard you are expected to put in disciplined and dedicated effort.

As our part, we assure of our guidance and support.

With best wishes,

SYLLABUS

B.Sc., Final Year

Ancillary Paper – II
PROGRAMMING IN C AND C++

UNIT – 1:

Introduction to C – The character set in C – Identifiers and Keywords – Data Types – Constants – Variables – Declaration – Expressions – Various Types of operators – Data Input/output and control statements – Preliminaries – single character input and output – Entering input data – writing output data – The gets and puts functions – Branching – Looping – Nested control structures – Switch – Break – Continue Goto.

UNIT – 2:

Function definition – Accessing a function – Function prototypes – Passing argument to function – Recursion – Library function – Macros – the C preprocessor – Storage classes – Automatic variables – Global variable – Static variable – Register variables – Multiple programming – Bitwise operations.

UNIT – 3:

Defining and processing of array – Passing arrays of functions – Multi dimensional arrays and strings.

UNIT – 4:

Defining a Structure – Processing a Structure – Structure and Functions – Self-referential Structures – Bit Field – Unions – Enumerations.

UNIT – 5:

Opening and Closing of Data files – Creating a Data File – Processing a data File – Unformatted data file – Command Line parameter.

UNIT – 6:

Principles of OOP – Software evolution – basic concepts of OOP – benefits of OOP – Object Oriented Languages – C++ tokens – keywords – identifiers – variables – operators manipulators – expressions and control structure in C++.

UNIT – 7:

Functions in C++ Main function – Function prototyping – Call by reference – return by reference function overloading – Friend and virtual function.

UNIT – 8:

Class and Objects – Constructors and Destructors – Polymorphism.

UNIT – 9:

Operator overloading and type conversion – Exceptions.

UNIT – 10:

Inheritance – various types of inheritance and Templates

Text Books:

For Units 1 to 5

1. "Progromming in C" by E.Balagurusamy 3rd edition, Tata McGraw Hills Rer Sachum's Series

For Units 6 to 10

2. Text Book: "Programming in C++" by E.Balagurusamy, 3rd edition Tata McGraw Hills Publications, 2005

SCHEME OF LESSONS

PRGRAMMING IN C AND C++

Table of Contents

SPACE FOR HINT

UNIT - 1

1. Introduction to C

C is a general-purpose programming language. It has been closely associated with the UNIX operating system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a ``system programming language'' because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 for the first UNIX system on the DEC PDP-7.

BCPL and B are ``typeless'' languages. By contrast, C provides a variety of data types. The fundamental types are characters, and integers and floating point numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures and unions. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement. Pointers provide for machine-independent address arithmetic.

C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (if-else), selecting one of a set of possible values (switch), looping with the termination test at the top (while, for) or at the bottom (do), and early loop exit (break).

Functions may return values of basic types, structures, unions, or pointers. Any function may be called recursively. Local variables are typically ``automatic'', or created anew with each invocation. Function definitions may not be nested but variables may be declared in a block-structured fashion. The functions of a C program may exist in separate source files that are compiled separately. Variables may be internal to a function, external but known only within a single source file, or visible to the entire program.

A preprocessing step performs macro substitution on program text, inclusion of other source files, and conditional compilation.

C is a relatively ``low-level'' language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

C provides no operations to deal directly with composite objects such as character strings, sets, lists or arrays. There are no operations that manipulate an entire array or string, although structures may be copied as a unit. The language does not define any storage allocation facility other than static definition and the stack discipline provided by the local variables of functions; there is no heap or garbage collection.

Finally, C itself provides no input/output facilities; there are no READ or WRITE statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly called functions. Most C implementations have included a reasonably standard collection of such functions.

Similarly, C offers only straightforward, single-thread control flow: tests, loops, grouping, and subprograms, but not multiprogramming, parallel operations, synchronization, or co-routines.

Although the absence of some of these features may seem like a grave deficiency, keeping the language down to modest size has real benefits. Since C is relatively small, it can be described in small space, and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language.

For many years, the definition of C was the reference manual in the first edition of The C Programming Language. In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or ``ANSI C", was completed in late 1988. Most of the features of the standard are already supported by modern compilers.

The standard is based on the original reference manual. The language is relatively little changed; one of the goals of the standard was to make sure that most existing programs would remain valid, or, failing that, that compilers could produce warnings of new behavior.

For most programmers, the most important change is the new syntax for declaring and defining functions. A function declaration can now include a description of the arguments of the function; the definition syntax changes to match. This extra information makes it much easier for compilers to detect errors caused by mismatched arguments; in our experience, it is a very useful addition to the language.

There are other small-scale language changes. Structure assignment and enumerations, which had been widely available, are now officially part of the language. Floating-point computations may now be done in single precision. The properties of arithmetic, especially for unsigned types, are clarified. The preprocessor is more elaborate. Most of these changes will have only minor effects on most programmers.

SPACE FOR HINT

SPACE FOR HINT

A second significant contribution of the standard is the definition of a library to accompany C. It specifies functions for accessing the operating system (for instance, to read and write files), formatted input and output, memory allocation, string manipulation, and the like. A collection of standard headers provides uniform access to declarations of functions in data types. Programs that use this library to interact with a host system are assured of compatible behavior. Most of the library is closely modeled on the ``standard I/O library" of the UNIX system. This library was described in the first edition, and has been widely used on other systems as well. Again, most programmers will not see much change.

Because the data types and control structures provided by C are supported directly by most computers, the run-time library required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can be avoided if they are not needed. Most can be written in C, and except for the operating system details they conceal, are themselves portable.

Although C matches the capabilities of many computers, it is independent of any particular machine architecture. With a little care it is easy to write portable programs, that is, programs that can be run without change on a variety of hardware. The standard makes portability issues explicit, and prescribes a set of constants that characterize the machine on which the program is run.

C is not a strongly-typed language, but as it has evolved, its type-checking has been strengthened. The original definition of C frowned on, but permitted, the interchange of pointers and integers; this has long since been eliminated, and the standard now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new function declarations are another step in this direction. Compilers will warn of most type errors, and there is no automatic conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.

C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better. Nonetheless, C has proven to be an extremely effective and expressive language for a wide variety of programming applications.

Why Use C?

In today's world of computer programming, there are many high-level languages to choose from, such as C, Pascal, BASIC, and Java. These are all excellent languages suited for most programming tasks. Even so, there are several reasons why many computer professionals feel that C is at the top of the list:

C is a powerful and flexible language. What you can accomplish with C is limited only by your imagination. The language itself places no constraints on you.

SPACE FOR HINT

C is used for projects as diverse as operating systems, word processors, graphics, spreadsheets, and even compilers for other languages.

C is a popular language preferred by professional programmers. As a result, a wide variety of C compilers and helpful accessories are available.

C is a portable language. Portable means that a C program written for one computer system (an IBM PC, for example) can be compiled and run on another system (a DEC VAX system, perhaps) with little or no modification. Portability is enhanced by the ANSI standard for C, the set of rules for C compilers.

C is a language of few words, containing only a handful of terms, called keywords, which serve as the base on which the language's functionality is built. You might think that a language with more keywords (sometimes called reserved words) would be more powerful. This isn't true. As you program with C, you will find that it can be programmed to do any task.

C is modular. C code can (and should) be written in routines called functions. These functions can be reused in other applications or programs. By passing pieces of information to the functions, you can create useful, reusable code.

As these features show, C is an excellent choice for your first programming language. What about C++? You might have heard about C++ and the programming technique called object-oriented programming. Perhaps you're wondering what the differences are between C and C++ .

Not to worry! C++ is a superset of C, which means that C++ contains everything C does, plus new additions for object-oriented programming. If you do go on to learn C++, almost everything you learn about C will still apply to the C++ superset. In learning C, you are not only learning one of today's most powerful and popular programming languages, but you are also preparing yourself for object-oriented programming.

Preparing to Program

You should take certain steps when you're solving a problem. First, you must define the problem. If you don't know what the problem is, you can't find a solution! Once you know what the problem is, you can devise a plan to fix it. Once you have a plan, you can usually implement it. Once the plan is implemented, you must test the results to see whether

SPACE FOR HINT

the problem is solved. This same logic can be applied to many other areas, including programming.

When creating a program in C (or for that matter, a computer program in any language), you should follow a similar sequence of steps:

1. Determine the objective(s) of the program.

2. Determine the methods you want to use in writing the program.

3. Create the program to solve the problem.

4. Run the program to see the results.

An example of an objective (see step 1) might be to write a word processor or database program. A much simpler objective is to display your name on the screen. If you didn't have an objective, you wouldn't be writing a program, so you already have the first step done.

The second step is to determine the method you want to use to write the program. Do you need a computer program to solve the problem? What information needs to be tracked? What formulas will be used? During this step, you should try to determine what you need to know and in what order the solution should be implemented.

As an example, assume that someone asks you to write a program to determine the area inside a circle. Step 1 is complete, because you know your objective: determine the area inside a circle. Step 2 is to determine what you need to know to ascertain the area. In this example, assume that the user of the program will provide the radius of the circle. Knowing this, you can apply the formula pi*r2 to obtain the answer. Now you have the pieces you need, so you can continue to steps 3 and 4, which are called the Program Development Cycle.

The Program Development Cycle

The Program Development Cycle has its own steps. In the first step, you use an editor to create a disk file containing your source code. In the second step, you compile the source code to create an object file. In the third step, you link the compiled code to create an executable file. The fourth step is to run the program to see whether it works as originally planned.

Creating the Source Code

Source code is a series of statements or commands that are used to instruct the computer to perform your desired tasks. As mentioned, the first step in the Program Development Cycle is to enter source code into an editor. For example, here is a line of C source code:

printf("Hello, Mom!");

This statement instructs the computer to display the message Hello, Mom! on-screen. (For now, don't worry about how this statement works.)

SPACE FOR HINT

Using an Editor

Most compilers come with a built-in editor that can be used to enter source code; however, some don't. Consult your compiler manuals to see whether your compiler came with an editor. If it didn't, many alternative editors are available.

Most computer systems include a program that can be used as an editor. If you're using a UNIX system, you can use such editors as ed, ex, edit, emacs, or vi. If you're using Microsoft Windows, Notepad is available. If you're using MS/DOS 5.0 or later, you can use Edit. If you're using a version of DOS before 5.0, you can use Edlin. If you're using PC/DOS 6.0 or later, you can use E. If you're using OS/2, you can use the E and EPM editors.

Most word processors use special codes to format their documents. These codes can't be read correctly by other programs. The American Standard Code for Information Interchange (ASCII) has specified a standard text format that nearly any program, including C, can use. Many word processors, such as WordPerfect, AmiPro, Word, WordPad, and WordStar, are capable of saving source files in ASCII form (as a text file rather than a document file). When you want to save a word processor's file as an ASCII file, select the ASCII or text option when saving.

If none of these editors is what you want to use, you can always buy a different editor. There are packages, both commercial and shareware, that have been designed specifically for entering source code.

When you save a source file, you must give it a name. The name should describe what the program does. In addition, when you save C program source files, give the file a .C extension. Although you could give your source file any name and extension, .C is recognized as the appropriate extension to use.

Compiling the Source Code

Although you might be able to understand C source code (at least, after reading this book you will be able to), your computer can't. A computer requires digital, or binary, instructions in what is called machine language. Before your C program can run on a computer, it must be translated from source code to machine language. This translation, the second step in program development, is performed by a program called a compiler. The compiler takes your source code file as input and

SPACE FOR HINT

produces a disk file containing the machine language instructions that correspond to your source code statements. The machine language instructions created by the compiler are called object code, and the disk file containing them is called an object file.

Each compiler needs its own command to be used to create the object code. To compile, you typically use the command to run the compiler followed by the source filename. The following are examples of the commands issued to compile a source file called RADIUS.C using various DOS/Windows compilers:

Compiler	**Command**
Microsoft C	cl radius.c
Borland's Turbo C	tcc radius.c
Borland C	bcc radius.c
Zortec C	ztc radius.c

To compile RADIUS.C on a UNIX machine, use the following command:

cc radius.c

Consult the compiler manual to determine the exact command for your compiler.

If you're using a graphical development environment, compiling is even simpler. In most graphical environments, you can compile a program listing by selecting the compile icon or selecting something from a menu. Once the code is compiled, selecting the run icon or selecting something from a menu will execute the program.

You should check your compiler's manuals for specifics on compiling and running a program.

After you compile, you have an object file. If you look at a list of the files in the directory or folder in which you compiled, you should find a file that has the same name as your source file, but with an .OBJ (rather than a .C) extension. The .OBJ extension is recognized as an object file and is used by the linker. On UNIX systems, the compiler creates object files with an extension of .O instead of .OBJ.

Linking to Create an Executable File

One more step is required before you can run your program. Part of the C language is a function library that contains object code (code that has already been compiled) for predefined functions. A predefined function contains C code that has already been written and is supplied in a ready-to-use form with your compiler package.

The printf() function used in the previous example is a library function. These library functions perform frequently needed tasks, such as displaying information on-screen and reading data from disk files. If your program uses any of these functions (and hardly a program exists that doesn't use at least one), the object file produced when your source code was compiled must be combined with object code from the function library to create the final executable program. (Executable means that the program can be run, or executed, on your computer.) This process is called linking, and it's performed by a program called (you guessed it) a linker.

SPACE FOR HINT

Completing the Development Cycle

Once your program is compiled and linked to create an executable file, you can run it by entering its name at the system prompt or just like you would run any other program. If you run the program and receive results different from what you thought you would, you need to go back to the first step. You must identify what caused the problem and correct it in the source code. When you make a change to the source code, you need to recompile and relink the program to create a corrected version of the executable file. You keep following this cycle until you get the program to execute exactly as you intended.

One final note on compiling and linking: Although compiling and linking are mentioned as two separate steps, many compilers, such as the DOS compilers mentioned earlier, do both as one step. Regardless of the method by which compiling and linking are accomplished, understand that these two processes, even when done with one command, are two separate actions.

The C Development Cycle

Step 1

Use an editor to write your source code. By tradition, C source code files have the extension .C (for example, MYPROG.C, DATABASE.C, and so on).

Step 2

Compile the program using a compiler. If the compiler doesn't find any errors in the program, it produces an object file. The compiler produces object files with an .OBJ extension and the same name as the source code file (for example, MYPROG.C compiles to MYPROG.OBJ). If the compiler finds errors, it reports them. You must return to step 1 to make corrections in your source code.

Step 3

SPACE FOR HINT

Link the program using a linker. If no errors occur, the linker produces an executable program located in a disk file with an .EXE extension and the same name as the object file (for example, MYPROG.OBJ is linked to create MYPROG.EXE).

Step 4

Execute the program. You should test to determine whether it functions properly. If not, start again with step 1 and make modifications and additions to your source code.

Your First C Program

You're probably eager to try your first program in C. To help you become familiar with your compiler, here's a quick program for you to work through. You might not understand everything at this point, but you should get a feel for the process of writing, compiling, and running a real C program.

This demonstration uses a program named HELLO.C, which does nothing more than display the words Hello, World! on-screen. This program, a traditional introduction to C programming, is a good one for you to learn. The source code for HELLO.C is given below When you type in this listing, you won't include the line numbers or colons.

HELLO.C.

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     printf("Hello, World!\n");
6:     return 0;
7: }
```

Entering and Compiling HELLO.C

To enter and compile the HELLO.C program, follow these steps:

1. Make active the directory your C programs are in and start your editor. As mentioned previously, any text editor can be used, but most C compilers (such as Borland's Turbo C++ and Microsoft's Visual C/C++) come with an integrated development environment (IDE) that lets you enter, compile, and link your programs in one convenient setting. Check the manuals to see whether your compiler has an IDE available.

2. Use the keyboard to type the HELLO.C source code exactly as shown in Listing 1.1. Press Enter at the end of each line.

NOTE: Don't enter the line numbers or colons. These are for reference only.

SPACE FOR HINT

3. Save the source code. You should name the file HELLO.C.

4. Verify that HELLO.C is on disk by listing the files in the directory or folder. You should see HELLO.C within this listing.

5. Compile and link HELLO.C. Execute the appropriate command specified by your compiler's manuals. You should get a message stating that there were no errors or warnings.

6. Check the compiler messages. If you receive no errors or warnings, everything should be okay.

If you made an error typing the program, the compiler will catch it and display an error message. For example, if you misspelled the word printf as prntf, you would see a message similar to the following:

Error: undefined symbols:_prntf in hello.c (hello.OBJ)

7. Go back to step 2 if this or any other error message is displayed. Open the HELLO.C file in your editor. Compare your file's contents carefully with Listing 1.1, make any necessary corrections, and continue with step 3.

8. Your first C program should now be compiled and ready to run. If you display a directory listing of all files named HELLO (with any extension), you should see the following:

HELLO.C, the source code file you created with your editor

HELLO.OBJ or HELLO.O, which contains the object code for HELLO.C

HELLO.EXE, the executable program created when you compiled and linked HELLO.C

9. To execute, or run, HELLO.EXE, simply enter hello. The message Hello, World! is displayed on-screen.

Congratulations! You have just entered, compiled, and run your first C program. Admittedly, HELLO.C is a simple program that doesn't do anything useful, but it's a start. In fact, most of today's expert C programmers started learning C in this same way--by compiling HELLO.C--so you're in good company.

Compilation Errors

SPACE FOR HINT

A compilation error occurs when the compiler finds something in the zzany of a dozen other things can cause the compiler to choke. Fortunately, modern compilers don't just choke; they tell you what they're choking on and where it is! This makes it easier to find and correct errors in your source code.

This point can be illustrated by introducing a deliberate error into HELLO.C. If you worked through that example (and you should have), you now have a copy of HELLO.C on your disk. Using your editor, move the cursor to the end of the line containing the call to printf(), and erase the terminating semicolon. HELLO.C should now look like Listing 1.2.

HELLO.C with an error.

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     printf("Hello, World!")
6:     return 0;
7: }
```

Next, save the file. You're now ready to compile it. Do so by entering the command for your compiler. Because of the error you introduced, the compilation is not completed. Rather, the compiler displays a message similar to the following:

hello.c(6) : Error: `;' expected

Looking at this line, you can see that it has three parts:

hello.c The name of the file where the error was found

(6) : The line number where the error was found

Error: `;' expected A description of the error

This message is quite informative, telling you that in line 6 of HELLO.C the compiler expected to find a semicolon but didn't. However, you know that the semicolon was actually omitted from line 5, so there is a discrepancy. You're faced with the puzzle of why the compiler reports an error in line 6 when, in fact, a semicolon was omitted from line 5. The answer lies in the fact that C doesn't care about things like breaks between lines. The semicolon that belongs after the printf() statement could have been placed on the next line (although doing so would be bad programming practice). Only after encountering the next command (return) in line 6 is the compiler sure that the semicolon is missing. Therefore, the compiler reports that the error is in line 6.

This points out an undeniable fact about C compilers and error messages. Although the compiler is very clever about detecting and localizing errors, it's no Einstein. Using your knowledge of the C language, you must interpret the compiler's messages and determine the actual location of any errors that are reported. They are often found on the line reported by the compiler, but if not, they are almost always on the preceding line. You might have a bit of trouble finding errors at first, but you should soon get better at it.

SPACE FOR HINT

NOTE: The errors reported might differ depending on the compiler. In most cases, the error message should give you an idea of what or where the problem is.

Before leaving this topic, let's look at another example of a compilation error. Load HELLO.C into your editor again and make the following changes:

1. Replace the semicolon at the end of line 5.

2. Delete the double quotation mark just before the word Hello.

Save the file to disk and compile the program again. This time, the compiler should display error messages similar to the following:

```
hello.c(5) : Error: undefined identifier `Hello'
hello.c(7) : Lexical error: unterminated string
Lexical error: unterminated string
Lexical error: unterminated string
Fatal error: premature end of source file
```

The first error message finds the error correctly, locating it in line 5 at the word Hello. The error message undefined identifier means that the compiler doesn't know what to make of the word Hello, because it is no longer enclosed in quotes. However, what about the other four errors that are reported? These errors, the meaning of which you don't need to worry about now, illustrate the fact that a single error in a C program can sometimes cause multiple error messages.

The lesson to learn from all this is as follows: If the compiler reports multiple errors, and you can find only one, go ahead and fix that error and recompile. You might find that your single correction is all that's needed, and the program will compile without errors.

Linker Error Messages

Linker errors are relatively rare and usually result from misspelling the name of a C library function. In this case, you get an Error: undefined symbols: error message, followed by the misspelled name (preceded by an underscore). Once you correct the spelling, the problem should go away.

SPACE FOR HINT

Summary

After reading this chapter, you should feel confident that selecting C as your programming language is a wise choice. C offers an unparalleled combination of power, popularity, and portability. These factors, together with C's close relationship to the C++ object-oriented language as well as Java, make C unbeatable.

This chapter explained the various steps involved in writing a C program--the process known as program development. You should have a clear grasp of the edit-compile-link-test cycle, as well as the tools to use for each step.

Errors are an unavoidable part of program development. Your C compiler detects errors in your source code and displays an error message, giving both the nature and the location of the error. Using this information, you can edit your source code to correct the error. Remember, however, that the compiler can't always accurately report the nature and location of an error. Sometimes you need to use your knowledge of C to track down exactly what is causing a given error message.

1.1 The Character set in C : Identifier And Keywords

1.1.1 Identifier - Variable

A variable is a sequence of program code with a name *(also called its identifier)*. A name or identifier in C can be anything from a single letter to a word. The name of a variable must begin with an alphabetic letter or the underscore '_' character but the other characters in the name can be chosen from the following groups:

a .. z (any letter from a to z)

A .. Z (any letter from A to Z)

0 .. 9 (any digit from 0 to 9)

_ (the underscore character)

Some examples of valid variable names are:

a

total

Out_of_Memory

VAR

integer etc...

SPACE FOR HINT

1.1.2 C Keywords

The C language reserves certain words that have special meanings to the language. Those reserved words are sometimes called C keywords. You should not use the C keywords as variable, constant, or function names in your program. The following are the 32 reserved C keywords:

Keyword	**Description**
auto	Storage class specifier
break	Statement
case	Statement
char	Type specifier
const	Storage class modifier
continue	Statement
default	Label
do	Statement
double	Type specifier
else	Statement
enum	Type specifier
extern	Storage class specifier
float	Type specifier
for	Statement
goto	Statement
if	Statement
int	Type specifier
long	Type specifier
register	Storage class specifier
return	Statement
short	Type specifier
signed	Type specifier
sizeof	Operator
static	Storage class specifier
struct	Type specifier
switch	Statement
typedef	Statement
union	Type specifier
unsigned	Type specifier
void	Type specifier
volatile	Storage class modifier
while	Statement

Note that all C keywords are written in lowercase letters. C is a case-sensitive language. Therefore, int, as shown in the list here, is considered as a C keyword, but INT is not.

SPACE FOR HINT

1.1.3 Data Types

In C variables do not only have names: they also have types. The type of a variable conveys to the compiler what sort of data will be stored in it. In C.we specify the types of variables in their declarations. This serves two purposes:
z
• It gives a compiler precise information about the amount of memory that will have to be given over to a variable when a program is finally run and what sort of arithmetic will have to be used on it (e.g. integer only or floating point or none).

• It provides the compiler with a list of the variables in a convenient place so that it can cross check names and types for any errors. There is a lot of different possible types in C. In fact it is possible for us to define our own, but there is no need to do this right away: there are some basic types which are provided by C ready for use. The names of these types are all reserved words in C and they are summarized as follows:

char	A single ASCII character
short	A short integer (usually 16-bits)
short int	A short integer
int	A standard integer (usually 32-bits)
long	A long integer
long int	A long integer (usually 32-bits, but increasingly 64 bits)
float A	floating point or real number (short)
long float	a long floating point number
double	A long floating point number

There is some repetition in these words. In addition to the above, the word unsigned can also be placed in front of any of these types. Unsigned means that only positive or zero values can be used. (i.e. there is no minus sign). The advantage of using this kind of variable is that storing a minus sign
takes up some memory, so that if no minus sign is present, larger numbers can be stored in the same kind of variable. The ANSI standard also allows the word signed to be placed in front of any of these types, so indicate the opposite of unsigned. On some systems variables are signed by default, whereas on others they are not.

Individual Types

char

A character type is a variable which can store a single ASCII character. Groups of char form strings. In C single characters are written enclosed

SPACE FOR HINT

by single quotes, e.g. 'c'! (This is in contrast to strings of many characters which use double quotes, e.g. "string") For instance, if ch is the name of a character:

char ch;

ch = 'a';
would give ch the value of the character a. The same effect can also be achieved by writing:

char ch = 'a';

A character can be any ASCII character, printable or not printable from values -128 to 127. (But only 0 to 127 are used.) Control characters i.e. non printable characters are put into programs by using a backslash \ and a special character or number. The characters and their meanings are:

'\b' backspace BS

'\f' form feed FF (also clear screen)

'\n' new line NL (like pressing return)

'\r' carriage return CR (cursor to start of line)

'\t' horizontal tab HT

'\v' vertical tab (not all versions)

'\"' double quotes (not all versions)

'\'' single quote character '

'\\' backslash character \

'\ddd' character ddd where ddd is an ASCII code given in octal or base 8

'\xddd' character ddd where ddd is an ASCII code given in hexadecimal

or base 16

```
#include <stdio.h>
main ()
{
printf ("Beep! \7 \n");
printf ("ch = \'a\' \n");
printf (" <- Start of this line!! \r");
}
```

The output of this program is:
Beep! (and the BELL sound)

SPACE FOR HINT

ch = 'a'

Integers

Whole numbers

There are five integer types in C and they are called char, int, long, long long and short. The difference between these is the size of the integer which either can hold and the amount of storage required for them. The sizes of these objects depend on the operating system of the computer. Even different flavours of Unix can have varying sizes for these objects. Usually, the two to remember are int and short. int means a 'normal' integer and short means a 'short' one, not that that tells us much. On a typical 32 bit microcomputer the size of these integers is the following:

Type	Bits	Possible Values
short	16	-32768 to 32767
unsigned short	16	0 to 65535
int	32	-2147483648 to 2147483647
long	32	(ditto)
unsigned int	32	0 to 4294967295
long long	64	-9e18 to + 8e18

Increasingly though, 64 bit operating systems are appearing and long integers are 64 bits long. You should always check these values. Some mainframe operating systems are completely 64 bit, e.g. Unicos has no 32 bit values.

Variables are declared in the usual way:

```
int i,j;

i = j = 0;
```

or

```
short i=0,j=0;
```

Floating Point

There are also long and short floating point numbers in C. All the mathematical functions which C can use require double or long float arguments Assigning variables to one another 43 so it is common to use the type float for storage only of small floating point numbers and to use double elsewhere. (This not always true since the C 'cast' operator

allows temporary conversions to be made.) On a typical 32 bit implementation the different types would be organized as follows:

SPACE FOR HINT

Type	Bits	Possible Values
float	32	+/- 10E-37 to +/- 10E38
double	64	+/- 10E-307 to +/- 10E308
long float	32	(ditto)

Typical declarations:

```
float x,y,z;

x = 0.1;

y = 2.456E5

z = 0;

double bignum,smallnum;

bignum = 2.36E208;

smallnum = 3.2E-300;
```

1.1.4 VARIABLES

Choosing Variables

The sort of procedure that you would adopt when choosing variable names is something like the following:

- Decide what a variable is for and what type it needs to be.
- Choose a sensible name for the variable.
- Decide where the variable is allowed to exist.
- Declare that name to be a variable of the chosen type.

Some local variables are only used temporarily, for controlling loops for instance. It is common to give these short names (single characters). A good habit to adopt is to keep to a consistent practice when using these variables. A common one, for instance is to use the letters:

```
int i,j,k;
```

to be integer type variables used for counting. (There is not particular reason why this should be; it is just common practice.) Other integer values should have more meaningful names. Similarly names like:

SPACE FOR HINT

```
double x,y,z;
```

tend to make one think of floating point numbers.

Assigning variables to one another

Variables can be assigned to numbers:

```
var = 10;
```

and assigned to each other:

```
var1 = var2;
```

In either case the objects on either side of the = symbol must be of the same type. It is possible (though not usually sensible) to assign a floating point number to a character for instance.
So int a, b = 1;

```
a = b;
```
is a valid statement, and:

```
float x = 1.4;

char ch;

ch = x;
```

is a valid statement, since the truncated value 1 can be assigned to ch. This is a questionable practice though. It is unclear why anyone would choose to do this. Numerical values and characters will interconvert because characters are stored by their ASCII codes (which are integers!) Thus the following will work:

```
int i;

char ch = 'A';

i = ch;

printf ("The ASCII code of %c is %d",ch,i);
```

The result of this would be:

The ASCII code of A is 65

Declarations and Initialization

When a variable is declared in C, the language allows a neat piece of syntax which means that variables can be declared and assigned a value in one go. This is no more efficient than doing it in two stages, but it is sometimes tidier. The following:

SPACE FOR HINT

```
int i = 0;
char ch = 'a';
```

are equivalent to the more longwinded

```
int i;
char ch;
i = 0;
ch = 'a';
```

This is called initialization of the variables. C always allows the programmer to write declarations/initializers in this way, but it is not always desirable to do so. If there are just one or two declarations then this initialization method can make a program neat and tidy. If there are many, then it is better to initialize separately, as in the second case. A lot means when it starts to look as though there are too many. It makes no odds to the compiler, nor (ideally) to the final code whether the first or second method is used. It is only for tidiness that this is allowed.

To declare a variable in a C program one writes the type followed by a list of variable names which are to be treated as being that type:

```
typename variablename1,..,..,variablenameN;
```

For example:

```
int i,j;

char ch;

double x,y,z,fred;

unsigned long int Name_of_Variable;
```

Failing to declare a variable is more risky than passing through customs and failing to declare your six tonnes of Swiss chocolate. A compiler is markedly more efficient than a customs officer: it will catch a missing declaration every time and will terminate a compiling session whilst complaining
bitterly, often with a host of messages, one for each use of the undeclared variable.

Where to declare things

There are two kinds of place in which declarations can be made,

1. One place is outside all of the functions. That is, in the space between function definitions. (After the #include lines, for example.) Variables declared here are called global variables. There are also called static and external variables in special cases.)

SPACE FOR HINT

```
#include <stdio.h>
int globalinteger; /* Here! outside {} */
float global_floating_point;
main ()
{
}
```

2. The other place where declarations can be made is following the opening brace, {}, of a block. Any block will do, as long as the declaration follows immediately after the opening brace. Variables of this kind only work inside their braces {} and are often called local variables. Another name
for them is automatic variables.

```
main ()
{ int a;
float x,y,z;
/* statements */
}
or
function ()
{ int i;
/* .... */
while (i < 10)
{ char ch;
int g;
/* ... */
}
}
```

1.2. Assignments, Expressions and Operators

An operator is something which takes one or more values and does some- thing useful with those values to produce a result. It operates on them. The terminology of operators is the following:

operator Something which operates on something.

SPACE FOR HINT

operand Each thing which is operated upon by an operator is called an operand.

operation The action which was carried out upon the operands by the operator!

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. C Operators can be classified into a number of categories. They include :

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and Decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

Most operators can be thought of as belonging to one of three groups, divided up arbitrarily according to what they do with their operands. These rough groupings are thought of as follows:

• Operators which produce new values from old ones. They make a result from their operands. e.g. +, the addition operator takes two numbers or two variables or a number and a variable and adds them together to give a new number.

• Operators which make comparisons. e.g. less than, equal to, greater than...

• Operators which produce new variable types: like the cast operator.

The majority of operators fall into the first group. In fact the second group is a subset of the first, in which the result of the operation is a Boolean value of either true of false.

1.2.1 Arithmetic Operators

The most common operators in any language are basic arithmetic operators. In C these are the following:

+ plus (unary)

- minus (force value to be negative)

+ addition

SPACE FOR HINT

AC-	subtraction
*	multiplication
/	floating point division
/	integer division "div"
%	integer remainder "mod"

1.2.2 Relational Operators

Six operators in C are for making logical comparisons. The six operators which compare values are:

==	is equal to
!=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

The values which they produce are called true and false. As words, "true" and "false" are not defined normally in C, but it is easy to define them as macros and they may well be defined in a library file:

```
#define TRUE 1
#define FALSE 0
```

Falsity is assumed to have the value zero in C and truth is represented by any non-zero value. These comparison operators are used for making decisions, but they are themselves operators and expressions can be built up with them.

```
1 == 1
```

has the value "true" (which could be anything except zero). The statement:

```
int i;
i = (1 == 2);
```

would be false, so i would be false. In other words, i would be zero.

Comparisons are often made in pairs or even in groups and linked together with words like OR and AND. For instance, some test might want to find out whether:

(A is greater than B) AND (A is greater than C)

SPACE FOR HINT

1.2.3 Logical Operators

C does not have words for these operations but gives symbols instead. The logical operators, as they are called, are as follows:

&& meaning logical AND

|| meaning logical OR

! meaning logical NOT

The statement which was written in words above could be translated as:

(A > B) && (A > C)

The statement:

(A is greater than B) AND (A is not greater than C)

translates to:

(A > B) && !(A > C)

The NOT operator always creates the logical opposite:

!true is false and !false is true. On or the other of these must be true. The question is therefore always true.

1.2.4 Assignment Operators

These operators would not be useful without a 'partner operator which could attach the values which they produce to variables. Perhaps the most important operator then is the assignment operator:

= assignment operator

The main assignment operators all follow this pattern:

+= add assign

-= subtract assign

*= multiply assign

/= divide (double) and (int) types

%= remainder (int) type only.

Example Listing

SPACE FOR HINT

```
#include <stdio.h>
main ()
{ int i;
printf ("Assignment Operators\n\n");
i = 10; /* Assignment */
printf("i = 10 : %d\n",i);
i++; /* i = i + 1 */
printf ("i++ : %d\n",i);
i += 5; /* i = i + 5 */
printf ("i += 5 : %d\n",i);
i--; /* i = i = 1 */
printf ("i-- : %d\n",i);
i -= 2; /* i = i - 2 */
printf ("i -= 2 : %d\n",i);
i *= 5; /* i = i * 5 */
printf ("i *= 5 :%d\n",i);
i /= 2; /* i = i / 2 */
printf ("i /= 2 : %d\n",i);
i %= 3; /* i = i % 3 */
printf ("i %%= 3 : %d\n",i);
}
```

Output

Assignment Operators

i = 10 : 10

i++ : 11

i += 5 : 16

i-- : 15

i -= 2 : 13

i *= 5 :65

i /= 2 : 32

i %= 3 : 2

This has been used extensively up to now. For example:

SPACE FOR HINT

```
double x,y;

x = 2.356;

y = x;

x = x + 2 + 3/5;
```

The assignment operator takes the value of whatever is on the right hand side of the '=' symbol and puts it into the variable on the left hand side. As usual there is some standard jargon for this, which is useful to know because compilers tend to use this when handing out error messages. The assignment operator can be summarized in the following way:

lvalue = expression;

lvalues on the other hand are simply names for memory locations: in other words variable names, or identifiers. The name comes from 'left values' meaning anything which can legally be written on the left hand side of an assignment.

This statement says no more than what has been said about assignments already: namely that it takes something on the right hand side and attaches it to whatever is on the left hand side of the '=' symbol. An expression is simply the name for any string of operators, variables and numbers. All of the following could be called **expressions:**

```
1 + 2 + 3
a + somefunction()
32 * x/3
i % 4
x
Output 113
1
(22 + 4*(function() + 2))
function () /* provided it returns a sensible value */
```

Example

```
main ()

{ int i;

p. intf ("Arithmetic Operators\n\n");
```

SPACE FOR HINT

```
i = 6;
printf ("i = 6, -i is : %d\n", -i);
printf ("int 1 + 2 = %d\n", 1 + 2);
printf ("int 5 - 1 = %d\n", 5 - 1);
printf ("int 5 * 2 = %d\n", 5 * 2);
printf ("\n9 div 4 = 2 remainder 1:\n");
printf ("int 9 / 4 = %d\n", 9 / 4);
printf ("int 9 % 4 = %d\n", 9 % 4);
printf ("double 9 / 4 = %f\n", 9.0 / 4.0);
}
```

Output

```
Arithmetic Operators
i = 6, -i is : -6
int 1 + 2 = 3
int 5 - 1 = 4
int 5 * 2 = 10
9 div 4 = 2 remainder 1:
int 9 / 4 = 2
int 9 4 = 1
double 9 / 4 = 2.250000
```

1.2.5 Increment (++) and Decrement Operators(--)

C has some special operators which cut down on the amount of typing involved in a program. This is a subject in which it becomes important to think in C and not in other languages. The simplest of these perhaps are the increment and decrement operators:

++ increment: add one to

-- decrement: subtract one from

SPACE FOR HINT

These attach to any variable of integer or floating point type. (character types too, with care.) They are used to simply add or subtract 1 from a variable. Normally, in other languages, this is accomplished by writing:

variable = variable + 1;

In C this would also be quite valid, but there is a much better way of doing this:

variable++; or

++variable;

would do the same thing more neatly. Similarly:

variable = variable - 1; is equivalent to variable--; or --variable;

Notice particularly that these two operators can be placed in front or after the name of the variable.

1.2.6 The Conditional ? Operator

C allows a very powerful and convenient operator that can be used to replaced statements of the life then else from. The ternary operator pair?: takes the general from

Exp 1 ?Exp 2: Exp 3

Where Exp1, Exp2 and Exp3 are expressions.

The ? operator works like this: this is evaluated. If it is true, Exp2is evaluated and becomes the value of the expression. If Exp1 is false, Exp3 is evaluated and its value becomes the value of the expression.

For example consider

X = 10;

Y = x>9 ? 100 : 200;

In this example y will be the value 100. if x had been less than would have less then 9, y would have received the value 200.

Example 2

a = 10;
b = 15
x = (a > b) ? a : b:

in this example, x will be assigned the value of b. This can be achieved using the if...else statements as follows :

SPACE FOR HINT

```
x = a;
else
x = b;
```

1.2.7 Bitwise Operators

Unlike many other languages C support is a complete set of bit wise operators. Since C is designed to take the place of assembly language of for most programming tasks, C must support all operations that can be done assembler. Bit wise operations to the testing setting or shifting of the actual bits in an integer or character variables, these operations **may not be used on type float or double.**

& bitwise AND

| bitwise OR

^ bitwise exclusive OR

<< Bit shift left (a specified number or bit positions)

>> Bit shift right(a specified number of bit positions)

The meaning and the syntax of these operators is given below.

Bitwise operations are not to be confused with logical operations (&&, ||...) A bit pattern is made up of 0s and 1s and bitwise operators operate individually upon each bit in the operand. Every 0 or 1 undergoes the operations

Shift Operations

Imagine a bit pattern as being represented by the following group of boxes. Every box represents a bit; the numbers inside represent their values. The values written over the top are the common integer values which the whole group of bits would have, if they were interpreted collectively as an integer.

```
128 64 3216 8  4  2   1
-----------------------------
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 1
-----------------------------
```

Shift operators move whole bit patterns left or right by shunting them between boxes. The syntax of this operation is:

value << number of positions

value >> number of positions

So for example, using the boxed value (1) above:

1 << 1 would have the value 2, because the bit pattern would have been moved one place the the left:

SPACE FOR HINT

```
128 64 32 16 8  4  2  1
-------------------------------
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | = 2
-------------------------------
```

Similarly: 1 << 4 has the value 16 because the original bit pattern is moved by four places:

```
128 64 32 16 8  4  2  1
-------------------------------
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | = 16
-------------------------------
```

And: 6 << 2 == 12

```
128 64 32 16 8  4  2  1
-------------------------------
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | = 6
-------------------------------
```

Shift left 2 places:

```
128 64 32 16 8  4  2  1
-------------------------------
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | = 12
-------------------------------
```

Notice that every shift left multiplies by 2 and that every shift right would divide by two, integer wise. If a bit reaches the edge of the group of boxes then it falls out and is lost forever. So:

1 >> 1 == 0
2 >> 1 == 1
2 >> 2 == 0
n >> n == 0

1.2.8 Special Operators

1.2.8.1. The Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression. For example the statement

value = (x =10, y=5, x+y)

First assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 (ie. 10 + 5) to value. Since comma operator has he lowest precedence of

SPACE FOR HINT

all operators, the parantheses are necessary. Some applications of comma operator are :

In for loops

for(n=1, m=10,n <=m; n++, m++)

in while loops

while (c =getchar(), c!=10)

Exchanging values

t = x, x = y , y = t;

1.2.8.2. The sizeof Operator

The sizeof is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples

m = sizeof(sum);

n = sizeof(long int);

k = sizeof(235L);

The sizeof operator is normally used to determine the length of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Parentheses and Priority

Parentheses are used for forcing a priority over operators. If an expression is written out in an ambiguous way, such as:

a + b / 4 * 2

it is not clear what is meant by this. It could be interpreted in several ways:

((a + b) / 4) * 2

or

(a + b)/ (4 * 2)

or

a + (b/4) * 2

and so on. By using parentheses, any doubt about what the expression means is removed. Parentheses are said to have a higher priority than +

* or / because they are evaluated as "sealed capsules" before other operators can act on them. Putting parentheses in may remove the ambiguity of expressions, but it does not alter than fact that a + b / 4 * 2 is ambiguous. What will happen in this case? The answer is that the C compiler has a convention about the way in which expressions are evaluated: it is called operator precedence. The convention is that some operators are stronger than others and that the stronger ones will always be evaluated first. Otherwise, expressions like the one above are evaluated from left to right: so an expression will be dealt with from left to right unless a strong operator overrides this rule. Use parentheses to be sure.

SPACE FOR HINT

Summary of Operators and Precedence

The highest priority operators are listed first.

Operator	Operation	Evaluated.
()	parentheses	left to right
[]	square brackets	left to right
++	increment	right to left
--	decrement	right to left
(type)	cast operator	right to left
*	the contents of	right to left
&	the address of	right to left
-	unary minus	right to left
~	one's complement	right to left
!	logical NOT	right to left
*	multiply	left to right
/	divide	left to right
%	remainder (MOD)	left to right
+	add	left to right
-	subtract	left to right
>>	shift right	left to right
<<	shift left	left to right
>	is greater than	left to right
>=	greater than or equal to	left to right
<=	less than or equal to	left to right
<	less than	left to right

SPACE FOR HINT

==	is equal to	left to right
!=	is not equal to	left to right
&	bitwise AND	left to right
^	bitwise exclusive OR	left to right
|	bitwise inclusive OR	left to right
&&	logical AND	left to right
||	logical OR	left to right
=	assign	right to left
+=	add assign	right to left
-=	subtract assign	right to left
*=	multiply assign	right to left
/=	divide assign	right to left
%=	remainder assign	right to left
>>=	right shift assign	right to left
<<=	left shift assign	right to left
&=	AND assign	right to left
^=	exclusive OR assign	right to left
|=	inclusive OR assign	right to left

1.3 DATA INPUT AND OUTPUT CONTROL STATEMENTS

Remember that there are no built in function to perform I/O operations in C. instead all of these function are found in the C standard library.

In C all I/O is character oriented. This not includes reading and written to the console (that is the keyboard and screen), but to the disk file function as well. This differs from BASIC in which you can read and write bytes. As you have seen in the functions gets () and getnum () it is possible to write functions that read strings and numbers but these functions still use calls to the cha5acter oriented I/O functions.

Console I/O:

Console I/O refers to operations that occur at the key board and screen of your computer, while you have already used some of these function this sections will discuss and clarify some important aspects of their usage.

If you only know I/O in BASIC be warned that the I/O in C is completely different. In BASIC all console I/O is performed using high level built in functions. Although some BASIC do support the INKEY $ FOR retraining one character, it is not the primary form of console input.

SPACE FOR HINT

1.3.1. SINGLE CHARACTER INPUT AND OUTPUT

The getchar () and putchar () Functions :

The simplest of the console I/O functions are getchar () which reads a character from the standard input (usually the keyboard) and putchar () which prints a character to the standard output (usually the screen).

The getchar() function waits until a key is pressed and then returns its value generally getchar will also "echo" the character you to the screen automatically. This means that the character you type at the keyboard will be explicitly written to the screen in order to be displayed without being echoed. It will not automatically appear because there is no connection between the keyboard and the screen.

The putchar () function will write its character argument to the screen of your computer only if that argument is part of the character set that you computer can display.

Formatted console I/O:

The standard C library contains two function that perform formatted output and input; printf () and scanf(). Formatted I/O refers to the fact that these function may format the information under your direction.

You are already some what familiar with printf() the complement of printf () scanf(), allows the reading of various data types from the keyboard including characters strings and numbers. Both printf () and scanf () allow the mixing of data format and the use of for example field specifies and decimal points.

1.3.2 THE printf() FUNCTION:

You can think of printf () a s a combination of BASIC'S PRINT statement and its advanced print using statement. To summarize what you learned about printf () in chapter 2, the general form of printf () if

Printf ("control string ", arguments list.)

The control string consists of two types of items. The first type is made up of characters that will be printed on the screen,. The second type contains format commands that define the way the subsequent arguments are displayed. There must be exactly the same number of

SPACE FOR HINT

formal commands as there are arguments m and the format commands and the arguments are matched in order.

For example this printf () call

Printf ("hi %c %d, %s" , 'c', 10, "there!")

Display Hi c 10 there !

The format control codes of printf();

Printf() code	**format**
%e	a single character
%d	decimal
%e	scientific notation
	%f decimal floating point
%g	uses of %e or %f,
which ever is shorter	
%o	octal
%s	string of characters
%u	unsigned decimal
%x	hexadecimal

The format control codes may have modifiers that specify the field width, the number of decimal places, and left justify the filed flag.

An integer placed between the %sign and the format command acts as a minimum field width specifies this pads the output with blanks or zeros to ensure that it is at least a certain minimum length. If the string of number is greater than that minimum , it will be printed in full even fit over runs the minimum. The default padding is done with spaces. If you to pad with zeros, you place a 0 before the field width spitfire . for example %05d will add pad a number of less than 5 digits with 0's.

To specify the number if decimal places printed for a floating point number of place a decimal point followed by the number of decimal point you wish to display after the field width specified. For example %10.4f will display a number at least 10 characters wide the four decimal places, this method also works when you want to specify the maximum field length of the strings and integer values. For example. %5.7s will display a string that will able at least five characters long and will the characters will be truncated off the end.

By default all output is right justified if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force the information to be displayed on the left by placing a minus sign directly after the %for example %10.2f will left justify a floating point number with two decimal places in a 10 character wide field.

The modifier tells printf() that a long int data type follows.

SPACE FOR HINT

With printf () you can cannot output virtually an format of data you desire, as you can see in the examples in table .by writing examples of your own and checking their results you can see if you fully understand the process.

Printf () statement	**output**
("%-5.2f", 123.234)	[123. 23]
("%-5.2f", 3.234)	[3.23]
("%10s", "hello")	[hello]
("%10s", "hello")	[hello]
("%5.7s", "123456789")	[1234567]

1.3.3 THE scanf () FUNCTION

The function scanf() a general purposes input routines, allows you to read formatted data and automatically convert numeric information into integers and floats. For example it is much like the reverse of printf (). The general from of scanf() is

Scanf (control string, arguments list);

The control string is made up of input format codes, which are preceded by a % sign . these codes are called listed in table.

The format commands can use fields length modifiers that ate integer numbers placed between the % and the format command code. An * placed after the % will suppress the assignment and advance to the next input field.

The scanf () format codes :

Code	**Meaning**
c	Read a single character
d	Read a decimal integer
e	Read a floating point number
f	Read a floating point number
h	Read a short integer
o	Read a octal number
s	Read a string
x	Read a hexadecimal number

any other character in the control string will be matched and discarded.

The input character in the control must be separated by spaces or new lines, function marks like commas or semicolons do not catch as separators. As in printf () the scanf() format codes match the variables receiving the input in order.

SPACE FOR HINT

All the variables used to revive values through scanf () must be possessed by their addresses. This that all arguments, other than the control string, must be pointers to the variables that will receive input. Remember , this is Cs way or creating a "call of by reference". For example if you wish to read an integer it to the variable count, you would use this scanf() call;

scanf ("%d", &count);

Strings will be read into character arrays, and the array name, without any index is the address of the first element of the array. Thus, to read string to the character array address you would use.

scanf ('%s", address);

In this case, address is already a pointer and need not be preceded by the & operator.

The maximum field length modifier may be applied to the format codes. If you wish to read no more than 20 character into address, you would write

scanf ("%20s", addresses);

If the input stream were greater than 20 character a subsequent call to the input would begin where this call left off. For examples.

11000 Park Way Ave, apt 2110 B

had been entered as the response to the earlier scanf () call only the first 20 characters, or up to the p in apt, would have been placed in to address because of the maximum size specify. This means that the remaining 8 characters , t 2110 B have not yet been used. If you make another scanf () call such as .

scanf () ("%s", str);

then t 2110 B would be placed int str. However many micro computer operating systems would simply lose characters that were typed but not assigned to anything.only if the system supports buffered I/O will those characters remain for processing.

1.3.4 THE gets () and puts() FUNCTIONS:

On the next step in terms of complexity and power, are the functions getchar() and puts (). They enable you to read and write strings of characters at the console.

```
Main ()          /* case switchar */
```

SPACE FOR HINT

```
{
        char ch;
        do {
                ch = getchar ();
                if (islower (ch)) putchar (toupper (ch));
                else putchar (tolower (ch));
        {  while (ch ! = '.');  /* use a period to stops */
}
```

the gets() function returns a null terminus string in its character array argument., this means that when you use gets() you may type character at the key board until you strike a carriage return. Striking a carriage return places a null terminator as the end of string and gets() returns. The carriage return itself is not contained in the string, and it is impossible to use gets () to return a carriage return; gethchar () can do so, however, the gets () allows you to correct mistakes by using the BACKSPACE key prior to pressing RETURN.

The puts() functions writes its string argument to the screen. A puts() recognizes the same backslash codes as printf () such as \n for new line a call to its ()requires for less overhead than the same dell to printf () because puts () only outputs a string of characters to the screen. It cannot output members or do format conversions. Therefore puts() takes up less space and runs dasher than printf() when displaying strings. Although a few compilers have neglected this functions puts() is in most standard libraries. If you do not have a puts () available the simple one shown the here will work if you compare it to most printf () functions this puts() is several times smaller;

```
Puts (s);
Char *s;
{
        register int t;
        for (t=0; s [t]; ++t ) putchar (s[t]);
}
```

the puts() function is important when the code size is important. If a program does not require all of the function of printf() it is to your advantage not to use a large function like printf () when the simple puts() function will do. For example

```
Getnum ()
{
        char num [80], n.;
        do{
                gets (num);
                if (! Number (num)) {
                        puts ("must be number \n");
                        n=0;
                }
```

SPACE FOR HINT

```
                    else n=1;
            }while (!n);
            return (atoi (num));
}

number (S)
char *s:
{
            int t;
            for (t=0; s [t]; ++t if ( ! is digit (s[t]))return 0;
            return 1;
}
```

this version will force you to enter a number because the support function number (), verifies that all of the characters in the string num are digits. The use of puts() instead of printf() mean if the using getnum () does not drag in the large function printf(). Remember if an function in your program requires printf() it work be loaded at like time so it is important to use the smallest function in standard functions.

The simplest console I/O functions:

Functions	**operations**
getchar ()	reads a character from the key board
putchar ()	writes a character to the screen
gets()	reads a string from the keyboard
puts()	writes a string to the screen

The simplest function that perform console I/O operations are summarized in table.

1.4. Decision Making : Branching and Looping

We are often faced with this dilemma: a situation in which a decision has to be made. The ability to make decisions and to choose different options is very useful in programming.

For instance, one might want to implement the following ideas in different programs:

• If the user hits the jackpot, write some message to say so. "You've won the game!"

• If a bank balance is positive then print C for credit otherwise print D for debit.

If the user has typed in one of five things then do something special for each special case, otherwise do something else.

SPACE FOR HINT

C language possesses such decision-making capabilities by supporting the following statements.

1. **If** statement
2. **Switch** statement
3. Conditional operator statement
4. **goto** statement

1.4.1. Decision making with if statement

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are

1. Simple if statement
2. If...else statement
3. Nested if...else statement
4. else if ladder

1.4.1.1 Simple if statement

The first form of the if statement is an all or nothing choice. if some condition is satisfied, do what is in the braces, otherwise just skip what is in the braces.

Formally, this is written:

```
if (condition) statement;
```

or

```
if (condition)
{
compound statement
}
```

Notice that, as well as a single statement, a whole block of statements can be written under the if statement. In fact, there is an unwritten rule of thumb in C that wherever a single statement will do, a compound statement will do instead. A compound statement is a block of single statements enclosed by curly braces.

SPACE FOR HINT

A condition is usually some kind of comparison, like the ones discussed in the previous chapter. It must have a value which is either true or false (1 or 0) and it must be enclosed by the parenthesis (and). If the condition has the value 'true' then the statement or compound statement following the condition will be carried out, otherwise it will be ignored. Some of the following examples help to show this:

```
int i;
printf ("Type in an integer");
scanf ("%ld",&i);
if (i == 0)
{
printf ("The number was zero");
}
if (i > 0)
{
printf ("The number was positive");
}
if (i < 0)
{
printf ("The number was negative");
}
```

The same code could be written more briefly, but perhaps less consistently in the following way:

```
int i;
printf ("Type in an integer");
scanf ("%ld",&i);
if (i == 0) printf ("The number was zero");
if (i > 0) printf ("The number was positive");
if (i < 0) printf ("The number was negative");
```

that it makes if statements look the same as all other block statements and it makes them stand out clearly in the program text. This rule of thumb is only dropped in very simple examples like:

```
if (i == 0) i++;
```

The if statement alone allows only a very limited kind of decision: it makes do or don't decisions

SPACE FOR HINT

Example Listings

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
/*****************************************/
main ()
{ int i;
if (TRUE)
{
printf ("This is always printed");
}
if (FALSE)
{
printf ("This is never printed");
}
}

/* On board car computer. Works out the */
/* number of kilometers to the litre */
/* that the car is doing at present */
#include <stdio.h>
main ()

{
double fuel,distance;
FindValues (&fuel,&distance);
Report (fuel,distance);
}

FindValues (fuel,distance) /* from car */
 double *fuel,*distance;
{
/* how much fuel used since last check on values */
printf ("Enter fuel used");
scanf ("%lf",fuel);
/* distance travelled since last check on values */
printf ("Enter distance travelled");
scanf ("%lf",distance);
}

Report (fuel,distance) /* on dashboard */
double fuel,distance;

{ double kpl;
kpl = distance/fuel;
```

SPACE FOR HINT

```
printf ("fuel consumption: %2.11f",kpl);
printf (" kilometers per litre\n");

if (kpl <= 1)
{
printf ("Predict fuel leak or car");
printf (" needs a service\n");
}
if (distance > 500)
{
printf ("Remember to check tyres\n");
}
if (fuel > 30) /* Tank holds 40 l */
{
printf ("Fuel getting low: %s left\n",40-fuel);
}
}
```

1.4.1.2. if ... else statement

The 'if .. else' statement has the form:

if (condition) statement1; else statement2;

This is most often written in the compound statement form:

```
if (condition)
{
statements
}
else
{
statements
}
```

The 'if..else' statement is a two way branch: it means do one thing or the other. When it is executed, the condition is evaluated and if it has the value 'true' (i.e. not zero) then statement1 is executed. If the condition is 'false' (or zero) then statement2 is executed. The 'if..else' construction often saves an unnecessary test from having to be made.

For instance:

```
int i;
scanf ("%ld",i);
if (i > 0)
{
printf ("That number was positive!");
}
else
```

```
{
printf ("That number was negative or zero!");
}
```

SPACE FOR HINT

It is not necessary to test whether i was negative in the second block because it was implied by the 'if..else' structure. That is, that block would not have been executed unless i were NOT greater than zero.

1.4.1.3 Nested ifs and logic statement

Consider the following statements which decide upon the value of some variable i. Their purposes are exactly the same.

```
if ((i > 2) && (i < 4))
{
printf ("i is three");
}
```

or:

```
if (i > 2)
{
if (i < 4)
{
printf ("i is three");
}
}
```

Both of these test i for the same information, but they do it in different ways. The first method might been born out of the following sequence of thought: If i is greater than 2 and i is less than four, both at the same time, then i has to be 3.

The second method is more complicated. Think carefully. It says: If i is greater than 2, do what is in the curly braces. Inside these curly braces i is always greater than 2 because otherwise the program would never have arrived inside them. Now, if i is also less than 4, then do what is inside the new curly braces. Inside these curly braces i is always less than 4. But wait! The whole of the second test is held inside the "i is greater than 2" braces, which is a sealed capsule: nothing else can get in, so, if the program gets into the "i is less than 4" braces as well, then both facts must be true at the same time. There is only one integer which is bigger than 2 and less than 4 at the ame time: it is 3. So i is 3.

The aim of this demonstration is to show that there are two ways of making multiple decisions in C. Using the logical comparison operators &&, || (AND,OR) and so on.. several multiple tests can be made. In many cases though it is too difficult to think in terms of these operators and the sealed capsule idea begins to look attractive. This is another advantage of using the curly braces: it helps the programmer to see that

SPACE FOR HINT

if statements and 'if..else' statements are made up of sealed capsule parts. Once inside a sealed capsule

```
if (i > 2)
{
/* i is greater than 2 in here! */
}
else
{
/* i is not greater than 2 here! */
}
```

the programmer can rest assured that nothing illegal can get in. The block braces are like regions of grace: they cannot be penetrated by anything which does not satisfy the right conditions. This is an enourmous weight off the mind! The programmer can sit back and think: I have accepted that i is greater than 2 inside these braces, so I can stop worrying about that now. This is how programmers learn to think in a structured way.

Example Listing

```
#include <stdio.h>
main ()
{
int persnum,usernum,balance;
persnum = 7462;
balance = -12;
printf ("The Plastic Bank Corporation\n");
printf ("Please enter your personal number :");
usernum = getnumber();

if (usernum == 7462)
{
printf ("\nThe current state of your account\n");
printf ("is %d\n",balance);

if (balance < 0)
{
printf ("The account is overdrawn!\n");
}
}
else
{
printf ("This is not your account\n");
}
printf ("Have a splendid day! Thank you.\n");
}
/**************************************************/
getnumber () /* get a number from the user */
{ int num = 0;
```

```
scanf ("%d",&num);
if ((num > 9999) || (num <= 0))
{
printf ("That is not a valid number\n");
}
return (num);
}
```

SPACE FOR HINT

1.4.1.4 Stringing together else if ladder

What is the difference between the following programs? They both interpret some imaginary exam result in the same way. They both look identical when compiled and run. Why then are they different?

```
/*************************************************/

/* Program 1 */

/*************************************************/

 #include <stdio.h>
main ()

{

int result;

printf("Type in exam result");

scanf ("%d",&result);

if (result < 10)

{

printf ("That is poor");

}

if (result > 20)

{

printf ("You have passed.");

}

if (result > 70)

{

printf ("You got an A!");

}

}

/* end */


/*************************************************/
```

SPACE FOR HINT

```
/* Program 2 */
/*************************************************/
#include <stdio.h>
main ()
{ int result;
printf("Type in exam result");
scanf ("%d",&result);
if (result < 10)
{
printf ("That is poor");
}
else
{
if (result > 20)
{
printf ("You have passed.");
}
else
{
if (result > 70)
{
printf ("You got an A!");
}
}
}
}
```

The answer is that the second of these programs can be more efficient. This because it uses the else form of the if statement which in turn means that few things have to be calculated. Program one makes every single test, because the program meets every if statement, one after the other.

The second program does not necessarily do this however. The nested if statements make sure that the second two tests are only made if the first one failed. Similarly the third test is only performed if the first two failed. So the second program could end up doing a third of the work of the first program, in the best possible case. Nesting decisions like this

can be an efficient way of controlling long lists of decisions like those above. Nested loops make a program branch into lots of possible paths, but choosing one path would preclude others.

SPACE FOR HINT

1.4.2 switch: integers and characters

The switch construction is another way of making a program path branch into lots of different limbs. It can be used as a different way of writing a string of 'if .. else' statements, but it is more versatile than that and it only works for integers and character type values. It works like a kind of multi-way switch. (See the diagram.) The switch statement has the following form:

```
switch (int or char expression)
{
case constant : statement;
break; /* optional */
...
}
```

It has an expression which is evaluated and a number of constant 'cases' which are to be chosen from, each of which is followed by a statement or compound statement. An extra statement called break can also be incorporated into the block at any point. break is a reserved word.

The switch statement can be written more specifically for integers:

```
switch (integer value)
{
case 1: statement1;
break; /* optional line */
case 2: statement2;
break; /* optional line */
....
default: default statement
break; /* optional line */
}
```

When a switch statement is encountered, the expression in the parentheses is evaluated and the program checks to see whether the result of that expression matches any of the constants labeled with case. If a match is made (for instance, if the expression is evaluated to 23 and there is a statement beginning "case 23 : ...") execution will start just

SPACE FOR HINT

after that case statement and will carry on until either the closing brace } is encountered or a break statement is found. break is a handy way of jumping straight out of the switch block. One of the cases is called default. Statements which follow the default case are executed for all cases which are not specifically listed.

Switch is a way of choosing some action from a number of known instances.

Look at the following example.

Example Listing

```
/* switch .. case */
/* Morse code program. Enter a number and */
/* find out what it is in Morse code */
#include <stdio.h>
#define CODE 0
main ()
{ short digit;
printf ("Enter any digit in the range 0..9");
scanf ("%h",&digit);
if ((digit < 0) || (digit > 9))
{
printf ("Number was not in range 0..9");
return (CODE);
}
printf ("The Morse code of that digit is ");
Morse (digit);
}
/*************************************************/
Morse (digit) /* print out Morse code */
short digit;
{
switch (digit)
{
case 0 : printf ("-----");
break;
```

```
case 1 : printf (".----");
break;
case 2 : printf ("..---");
break;
case 3 : printf ("...--");
break;
case 4 : printf ("....-");
break;
case 5 : printf (".....");
break;
case 6 : printf ("-....");
break;
case 7 : printf ("--...");
break;
case 8 : printf ("---..");
break;
case 9 : printf ("----.");
}
}
```

SPACE FOR HINT

The program selects one of the printf statements using a switch construction. At every case in the switch, a break statement is used. This causes control to jump straight out of the switch statement to its closing brace }. If break were not included it would go right on executing the statements to the end, testing the cases in turn. break this gives a way of jumping out of a switch quickly. There might be cases where it is not necessary or not desirable to jump out of the switch immediately. Think of a function yes() which gets a character from the user and tests whether it was 'y' or 'Y'.

```
yes () /* A sloppy but simple function */
{
switch (getchar())
{
case 'y' :
case 'Y' : return TRUE
default : return FALSE
```

SPACE FOR HINT

```
}
}
```

If the character is either 'y' or 'Y' then the function meets the statement return TRUE. If there had been a break statement after case 'y' then control would not have been able to reach case 'Y' as well. The return statement does more than break out of switch, it breaks out of the whole function, so in this case break was not required. The default option ensures that whatever else the character is, the function returns false.

1.4.3 The Conditional ? Operator:

C allows a very powerful and convenient operator that can be used to replaced statements of the life then else from. The ternary operator pair?: takes the general from

Exp 1 ?Exp 2: Exp 3

Where Exp1, Exp2 and Exp3 are expressions.

The ? operator works like this: this is evaluated. If it is true, Exp2is evaluated and becomes the value of the expression. If Exp1 is false, Exp3 is evaluated and its value becomes the value of the expression.

For example consider

```
x = 10;

y = x > 9 ? 100 : 200;
```

In this example y will be the value 100. if x had been less than would have less then 9, y would have received the value 200.

Example 2

```
a = 10;
b = 15
x = ( a > b ) ? a : b:
```

in this example, x will be assigned the value of b. This can be achieved using the if...else statements as follows :

```
if (a > b)
x = a;
else
x = b;
```

SPACE FOR HINT

1.4.4 GOTO statements

C which has a rich set of control structures and allows additional control using break and continue there is little need for the goto, the chief concern most programmers have about the goto is the tendency it confuse a program and render it nearly unread however there are times when the has for the goto will actually clarify program for assembler you should least be introduced to the goto because replacement for assembler you should at labels code to a requiem a label to be very short and fast,.

The goto requires a label for operation. A label may be any name that starts with a letter or an undertake followed by either numbers single number or the underscore character the label name must be followed by a colon for example a loop from 1 to 100 could be written using goto like this.

```
x=1;
loop 1;
                x++ ;
                If (x<100) goto loop1;
```

A good for the goto is as a way to exit fro several layers of nesting. In this example.

```
For (.....) {
        For (.....) {
                While (.....){
                        If (.....) goto stop;
                        .
                        .
                        .
                }
        }
}
stop:
        printf(" error in program \n")'
```

eliminating the got to should cause a number if additional tests to be performed. A simple break statement would not work here because it would only exit from the innermost loop.

1.5. Loops

Controlling repetitive processes. Nesting loops Decisions can also be used to make up loops. They allow the programmer to build a sequence of instructions which can be executed again and again, with some condition deciding when they will stop.

There are three kinds of loop in C. They are called:

SPACE FOR HINT

1. The while statement
2. The do statement
3. The for statement

These three loops offer a great amount of flexibility to programmers and can be used in some surprising ways!

1.5.1 while statement

The simplest of the three loops is the while loop. In common language while has a fairly obvious meaning: the while-loop has a condition:

```
while (condition)
{
statements;
}
```

and the statements in the curly braces are executed while the condition has the value "true" (1). There are dialects of English, however, in which Loops "while" does not have its commonplace meaning, so it is worthwhile explaining the steps which take place in a while loop.

The first important thing about this loop is that has a conditional expression (something like (a > b) etc...) which is evaluated every time the loop is executed by the computer. If the value of the expression is true, then it will carry on with the instructions in the curly braces.

If the expression evaluates to false (or 0) then the instructions in the braces are ignored and the entire while loop ends. The computer then moves onto the next statement in the program.

The second thing to notice about this loop is that the conditional expression comes at the start of the loop: this means that the condition is tested at the start of every 'pass', not at the end.

The reason that this is important is this: if the condition has the value false before the loop has been executed even once, the statements inside the braces will not get executed at all – not even once.

Example

It uses a while loop which is always true to repeat the process of getting a response from the user.

When the response is either yes or no it quits using the return function to jump right out of the loop.

SPACE FOR HINT

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
 main ()
{
printf ("Yes or no? (Y/N)\n");
if (yes())
{
printf ("YES!");
}
else
{
printf ("NO!");
}
}

 yes () /* get response Y/N query */
{
char getkey();
while (true)
{
switch (getkey())
{
case 'y' : case 'Y' : return (TRUE);
case 'n' : case 'N' : return (FALSE);
}
}
}

char getkey () /* get a character + RETURN */
{ char ch;
ch = getchar();
skipgarb();
}
```

SPACE FOR HINT

```
skipgarb ()
v
while (getchar() != '\n')
{
}
}
/* end */
```

Example Listing

This example listing prompts the user to type in a line of text and it counts all the spaces in that line. It quits when there is no more input left and printf out the number of spaces.

```
/* count all the spaces in an line of input */

#include <stdio.h>
main ()
{ char ch;
short count = 0;
printf ("Type in a line of text\n");
while ((ch = getchar()) != '\n')
{
if (ch == ' ')
{
count++;
}
}
printf ("Number of space = %d\n",count);
}
```

1.5.2 do..while statement

The do.. while loop has the form:

```
do
{
statements;
}
```

while (condition)

SPACE FOR HINT

Notice that the condition is at the end of this loop. This means that a do..while loop will always be executed at least once, before the test is made to determine whether it should continue. This is the only difference between while and do..while.

Example Listing

Here is an example of the use of a do..while loop. This program gets a line of input from the user and checks whether it contains a string marked out with "" quote marks. If a string is found, the program prints out the contents of the string only. A typical input line might be:

Once upon a time "Here we go round the..."what a terrible..

The output would then be:

Here we go round the...

If the string has only one quote mark then the error message 'string was not closed before end of line' will be printed.

```
#include <stdio.h>
main ()
{ char ch,skipstring();
do
{
if ((ch = getchar()) == '"')
{
printf ("The string was:\n");
ch = skipstring();
}
}
while (ch != '\n')
{
}
}

char skipstring () /* skip a string "..." */
{ char ch;
```

SPACE FOR HINT

```
do
{
ch = getchar();
putchar(ch);
if (ch == '\n')
{
printf ("\nString was not closed ");
printf ("before end of line\n");
break;
}
}
while (ch != '"')
{
}
return (ch);
}
```

1.5.3 for statement

The most interesting and also the most difficult of all the loops is the for loop. A for loop normally has the characteristic feature of controlling one particular variable, called the control variable. That variable is somehow associated with the loop.

For example it might be a variable which is used to count "for values from 0 to 10" or whatever.

The form of the for loop is:

```
for (statement1; condition; statement2)
{
}
```

For normal usage, these expressions have the following significance.

statement1

This is some kind of expression which initializes the control variable. This statement is only carried out once before the start of the loop. e.g. i = 0;

condition

This is a condition which behaves like the while loop. The condition is evaluated at the beginning of every loop and the loop is only carried out while this expression is true. e.g. i < 20;

statement2

This is some kind of expression for altering the value of the control variable.

```
for (x = 0; x <= 10; x += 0.5)
{
}
```

The result is that a C for loop often has the <= symbol in it. The for loop has plenty of uses. It could be used to find the sum of the first n natural numbers very simply:

```
sum = 0;
for (i = 0; i <= n; i++)
{
sum += i;
}
```

It generally finds itself useful in applications where a single variable has to be controlled in a well determined way.

Example Listing

This example program prints out all the primes numbers between 1 and the macro value maxint. Prime numbers are numbers which cannot be divided by any number except 1 without leaving a remainder.

```
#include <stdio.h>
#define MAXINT 500
#define TRUE 1
#define FALSE 0
main ()
{ int i;
for (i = 2; i <= MAXINT; i++)
{
if (prime(i))
{
```

SPACE FOR HINT

SPACE FOR HINT

```
printf ("%5d",i);
}
}
}
prime (i) /* check for a prime number */
int i;
{ int j;
for (j = 2; j <= i/2; j++)
{
if (i % j == 0)
{
return FALSE;
}
}
return TRUE;
}
```

The flexible for loop

The word 'statement' was chosen carefully, above, to describe what goes into a for loop. Look at the loop again:

```
for (statement1; condition; statement2)
{
.....
}
```

Statement really means what it says. C will accept any statement in the place of those above, including the empty statement. The while loop could be written as a for loop!

```
for (; condition; ) /* while ?? */
{
.....
}
```

Here there are two empty statements, which are just wasted. This flexibility can be put to better uses though. Consider the following loop:

```
for (x = 2; x <= 1000; x = x * x)
```

SPACE FOR HINT

```
{
....
}
```

This loop begins from 2 and each time the statements in the braces are executed x squares itself! Another odd looking loop is the following one:

```
for (ch = '*'; ch != '\n'; ch = getchar())
{
}
```

This could be used to make yet another different kind of skipgarb() function. The loop starts off by initializing ch with a star character. It checks that ch != '\n' (which it isn't, first time around) and proceeds with the loop. On each new pass, ch is reassigned by calling the function getchar().

It is also possible to combine several incremental commands in a loop:

```
for (i = 0, j=10; i < j; i++, j--)
{
printf("i = %d, j= %d\n",i,j);
}
```

Nested Loops

Like decisions, loops will also nest: that is, loops can be placed inside other loops. Although this feature will work with any loop at all, it is most commonly used with the for loop, because this is easiest to control. A for loop controls the number of times that a particular set of statements will be carried out. Another outer loop could be used to control the number of times that a whole loop is carried out.

To see the benefit of nesting loops, the example below shows how a square could be printed out using two printf statements and two loops.

```
#include <stdio.h>
#define SIZE 10
main ()
{ int i,j;
for (i = 1; i <= SIZE; i++)
{
for (j = 1; j <= SIZE; j++)
```

SPACE FOR HINT

```
{
printf("*");
}
printf ("\n");
}
}
```

The output of this program is a "kind of" square:

```
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
```

Quitting Loops

1.6. BREAK AND CONTINUE

C provides a simple way of jumping out of any of the three loops above at any stage, whether it has finished or not. The statement which performs this action is the same statement which was used to jump out of switch statements in last section.

```
break;
```

If this statement is encountered a loop will quit where it stands. For instance, an expensive way of assigning i to be 12 would be:

```
for (i = 1; i <= 20; i++)
{
if (i == 12)
{
break;
}
}
```

SPACE FOR HINT

Still another way of making skipgarb() would be to perform the following loop:

```
while (TRUE)
{
ch = getchar();
if (ch == '\n')
{
break;
}
}
```

Of course, another way to do this would be to use the return() statement, which jumps right out of a whole function. break only jumps out of the loop, so it is less drastic. As well as wanting to quit a loop, a programmer might want to hurry a loop on to the next pass: perhaps to avoid executing a lot of irrelevant statements, for instance. C gives a statement for this too, called:

continue;

When a continue statement is encountered, a loop will stop whatever it is doing and will go straight to the start of the next loop pass. This might be useful to avoid dividing by zero in a program:

```
for (i = -10; i <= 10; i++)
{
if (i == 0)
{
continue;
}
printf ("%d", 20/i);
}
```

1.7 GOTO:

C which has a rich set of control structures and allows additional control using break and continue there is little need for the goto, the chief concern most programmers have about the goto is the tendency it confuse a program and render it nearly unread however there are times when the has for the goto will actually clarify program for assembler

SPACE FOR HINT

you should least be introduced to the goto because replacement for assembler you should at labels code to a requiem a label to be very short and fast,.

The goto requires a label for operation. A label may be any name that starts with a letter or an undertake followed by either numbers single number or the underscore character the label name must be followed by a colon for example a loop from 1 to 100 could be written using gotc like this.

```
x=1;
loop 1;
                x++ ;
                If (x<100) goto loop1;
```

A good for the goto is as a way to exit fro several layers of nesting. I this example.

```
For (.....) {
        For (.....) {
                While (.....){
                        If (.....) goto stop;
                        .
                        .
                        .
                }
        }
}
stop:
        printf(" error in program \n")'
```

eliminating the got to should cause a number if additional tests to be performed. A simple break statement would not work here because it would only exit from the innermost loop.

C.Y.P

1. Explain about various data types in C

2. Explain about various types operators in detail.

3. Explain about single character input and output

4. Discuss about Branching.

5. Discuss about various Control Structure in detail.

Table of Contents

SPACE FOR HINT

UNIT – 2

2. INTRODUCTION

Functions

Functions are central to C programming and to the philosophy of C program design. This units covers user-defined functions, which, as the name implies, are functions that you, the programmer, create.

What a function is and what its parts are

About the advantages of structured programming with functions

How to create a function

How to declare local variables in a function

How to return a value from a function to the program

How to pass arguments to a function

What Is a Function?

A Function Defined

First the definition: A function is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program. Now let's look at the parts of this definition:

A function is named. Each function has a unique name. By using that name in another part of the program, you can execute the statements contained in the function. This is known as calling the function. A function can be called from within another function.

A function is independent. A function can perform its task without interference from or interfering with other parts of the program.

A function performs a specific task. This is the easy part of the definition. A task is a discrete job that your program must perform as part of its overall operation, such as sending a line of text to a printer, sorting an array into numerical order, or calculating a cube root.

A function can return a value to the calling program. When your program calls a function, the statements it contains are executed. If you want them to, these statements can pass information back to the calling program.

A Function Illustrated

SPACE FOR HINT

A program that uses a function to calculate the cube of a number.

```
1:  /* Demonstrates a simple function */
2:  #include <stdio.h>
3:
4:  long cube(long x);
5:
6:  long input, answer;
7:
8:  main()
9:  {
10:    printf("Enter an integer value: ");
11:    scanf("%d", &input);
12:    answer = cube(input);
13:    /* Note: %ld is the conversion specifier for */
14:    /* a long integer */
15:    printf("\nThe cube of %ld is %ld.\n", input, answer);
16:
17:    return 0;
18: }
19:
20: /* Function: cube() - Calculates the cubed value of a variable */
21: long cube(long x)
22: {
23:    long x_cubed;
24:
25:    x_cubed = x * x * x;
26:    return x_cubed;
27: }
```

```
Enter an integer value: 100
The cube of 100 is 1000000.
Enter an integer value: 9
The cube of 9 is 729.
Enter an integer value: 3
The cube of 3 is 27.
```

SPACE FOR HINT

How a Function Works

A C program doesn't execute the statements in a function until the function is called by another part of the program. When a function is called, the program can send the function information in the form of one or more arguments. An argument is program data needed by the function to perform its task. The statements in the function then execute, performing whatever task each was designed to do. When the function's statements have finished, execution passes back to the same location in the program that called the function. Functions can send information back to the program in the form of a return value.

Each time a function is called, execution passes to that function. When the function is finished, execution passes back to the place from which the function was called. A function can be called as many times as needed, and functions can be called in any order.

When a program calls a function, execution passes to the function and then back to the calling program.

You now know what a function is and the importance of functions.

2.1. Function Definition

```
return_type function_name( arg-type name-1,...,arg-type name-n)
{
   /* statements; */
}
```

A function prototype provides the compiler with a description of a function that will be defined at a later point in the program. The prototype includes a return type indicating the type of variable that the function will return. It also includes the function name, which should describe what the function does. The prototype also contains the variable types of the arguments (arg type) that will be passed to the function. Optionally, it can contain the names of the variables that will be passed. A prototype should always end with a semicolon.

A function definition is the actual function. The definition contains the code that will be executed. The first line of a function definition, called the function header, should be identical to the function prototype, with the exception of the semicolon. A function header shouldn't end with a semicolon. In addition, although the argument variable names were optional in the prototype, they must be included in the function header. Following the header is the function body, containing the statements that the function will perform. The function body should start with an opening bracket and end with a closing bracket. If the function return type is anything other than void, a return statement should be included, returning a value matching the return
type.

2.2. Function Prototype

SPACE FOR HINT

return_type function_name(arg-type name-1,...,arg-type name-n);

Function Prototype Examples

```
double squared( double number );
void print_report( int report_number );
int get_menu_choice( void );
```

Function Definition Examples

```
double squared( double number )          /* function header */
{                                        /* opening bracket */
   return(  number * number );           /* function body   */
}                                        /* closing bracket */

void print_report( int report_number )

   if( report_number == 1 )
      puts( "Printing Report 1" );
   else
      puts( "Not printing Report 1" );
}
```

Writing a Function

The first step in writing a function is knowing what you want the function to do. Once you know that, the actual mechanics of writing the function aren't particularly difficult.

The Function Header

The first line of every function is the function header, which has three components, each serving a specific function.

The Function Return Type

The function return type specifies the data type that the function returns to the calling program. The return type can be any of C's data types: char, int, long, float, or double. You can also define a function that doesn't return a value by using a return type of void. Here are some examples:

```
int func1(...)        /* Returns a type int.   */
float func2(...)      /* Returns a type float. */
void func3(...)       /* Returns nothing.      */
```

SPACE FOR HINT

The Function Name

You can name a function anything you like, as long as you follow the rules for C variable names. A function name must be unique (not assigned to any other function or variable). It's a good idea to assign a name that reflects what the function does.

The Parameter List

Many functions use arguments, which are values passed to the function when it's called. A function needs to know what kinds of arguments to expect--the data type of each argument. You can pass a function any of C's data types. Argument type information is provided in the function header by the parameter list.

For each argument that is passed to the function, the parameter list must contain one entry. This entry specifies the data type and the name of the parameter. For example

```
long cube(long x)
```

The parameter list consists of long x, specifying that this function takes one type long argument, represented by the parameter x. If there is more than one parameter, each must be separated by a comma. The function header

```
void func1(int x, float y, char z)
```

specifies a function with three arguments: a type int named x, a type float named y, and a type char named z. Some functions take no arguments, in which case the parameter list should consist of void, like this:

```
void func2(void)
```

NOTE: You do not place a semicolon at the end of a function header. If you mistakenly include one, the compiler will generate an error message.

Sometimes confusion arises about the distinction between a parameter and an argument. A parameter is an entry in a function header; it serves as a "placeholder" for an argument. A function's parameters are fixed; they do not change during program execution.

An argument is an actual value passed to the function by the calling program. Each time a function is called, it can be passed different arguments. In C, a function must be passed the same number and type of arguments each time it's called, but the argument values can be different. In the function, the argument is accessed by using the corresponding parameter name.

SPACE FOR HINT

An example will make this clearer. Here presents a very simple program with one function that is called twice.

The difference between arguments and parameters.

```
1:  /* Illustrates the difference between arguments and parameters. */
2:
3:  #include <stdio.h>
4:
5:  float x = 3.5, y = 65.11, z;
6:
7:  float half_of(float k);
8:
9:  main()
10: {
11:    /* In this call, x is the argument to half_of(). */
12:    z = half_of(x);
13:    printf("The value of z = %f\n", z);
14:
15:    /* In this call, y is the argument to half_of(). */
16:    z = half_of(y);
17:    printf("The value of z = %f\n", z);
18:
19:    return 0;
20: }
21:
22: float half_of(float k)
23: {
24:    /* k is the parameter. Each time half_of() is called, */
25:    /* k has the value that was passed as an argument. */
26:
27:    return (k/2);
28: }
```

```
The value of z = 1.750000
The value of z = 32.555000
```

The Function Body

The function body is enclosed in braces, and it immediately follows the function header. It's here that the real work is done. When a function is called, execution begins at the start of the function body and terminates (returns to the calling program) when a return statement is encountered or when execution reaches the closing brace.

Local Variables

You can declare variables within the body of a function. Variables declared in a function are called local variables. The term local means

SPACE FOR HINT

that the variables are private to that particular function and are distinct from other variables of the same name declared elsewhere in the program. This will be explained shortly; for now, you should learn how to declare local variables.

A local variable is declared like any other variable, using the same variable types and rules for names.Local variables can also be initialized when they are declared. You can declare any of C's variable types in a function. Here is an example of four local variables being declared within a function:

```
int func1(int y)
{
   int a, b = 10;
   float rate;
   double cost = 12.55;
   /* function code goes here... */
}
```

The preceding declarations create the local variables a, b, rate, and cost, which can be used by the code in the function. Note that the function parameters are considered to be variable declarations, so the variables, if any, in the function's parameter list also are available.

When you declare and use a variable in a function, it is totally separate and distinct from any other variables that are declared elsewhere in the program. This is true even if the variables have the same name. Listing 5.3 demonstrates this independence.

Listing : A demonstration of local variables.

```
1:  /* Demonstrates local variables. */
2:
3:  #include <stdio.h>
4:
5:  int x = 1, y = 2;
6:
7:  void demo(void);
8:
9:  main()
10: {
11:   printf("\nBefore calling demo(), x = %d and y = %d.", x, y);
12:   demo();
13:   printf("\nAfter calling demo(), x = %d and y = %d\n.", x, y);
14:
15:   return 0;
16: }
17:
18: void demo(void)
19: {
```

```
20:     /* Declare and initialize two local variables. */
21:
22:     int x = 88, y = 99;
23:
24:     /* Display their values. */
25:
26:     printf("\nWithin demo(), x = %d and y = %d.", x, y);
27: }
```

```
Before calling demo(), x = 1 and y = 2.
Within demo(), x = 88 and y = 99.
After calling demo(), x = 1 and y = 2.
```

SPACE FOR HINT

Returning a Value

To return a value from a function, you use the return keyword, followed by a C expression. When execution reaches a return statement, the expression is evaluated, and execution passes the value back to the calling program. The return value of the function is the value of the expression. Consider this function:

```
int func1(int var)
{
   int x;
   /* Function code goes here... */
   return x;
}
```

When this function is called, the statements in the function body execute up to the return statement. The return terminates the function and returns the value of x to the calling program. The expression that follows the return keyword can be any valid C expression.

A function can contain multiple return statements. The first return executed is the only one that has any effect. Multiple return statements can be an efficient way to return different values from a function, here is an example

Using multiple return statements in a function.

```
1:  /* Demonstrates using multiple return statements in a function. */
2:
3:  #include <stdio.h>
4:
5:  int x, y, z;
6:
7:  int larger_of( int , int );
8:
9:  main()
10: {
```

SPACE FOR HINT

```
11:     puts("Enter two different integer values: ");
12:     scanf("%d%d", &x, &y);
13:
14:     z = larger_of(x,y);
15:
16:     printf("\nThe larger value is %d.", z);
17:
18:     return 0;
19: }
20:
21: int larger_of( int a, int b)
22: {
23:    if (a > b)
24:       return a;
25:    else
26:       return b;
27: }
```

```
Enter two different integer values:
200 300
The larger value is 300.
Enter two different integer values:
300
200
The larger value is 300.
```

2.3 Passing Arguments to a Function

To pass arguments to a function, you list them in parentheses following the function name. The number of arguments and the type of each argument must match the parameters in the function header and prototype.

For example, if a function is defined to take two type int arguments, you must pass it exactly two int arguments--no more, no less--and no other type. If you try to pass a function an incorrect number and/or type of argument, the compiler will detect it, based on the information in the function prototype.

If the function takes multiple arguments, the arguments listed in the function call are assigned to the function parameters in order: the first argument to the first parameter, the second argument to the second parameter, and so on, as shown in Figure 5.5.

Each argument can be any valid C expression: a constant, a variable, a mathematical or logical expression, or even another function (one with a return value). For example, if half(), square(), and third() are all functions with return values, you could write

```
x = half(third(square(half(y))));
```

The program first calls half(), passing it y as an argument. When execution returns from half(), the program calls square(), passing half()'s return value as an argument. Next, third() is called with square()'s return value as the argument. Then, half() is called again, this time with third()'s return value as an argument. Finally, half()'s return value is assigned to the variable x. The following is an equivalent piece of code:

```
a = half(y);
b = square(a);
c = third(b);
x = half(c);
```

SPACE FOR HINT

Calling Functions

There are two ways to call a function. Any function can be called by simply using its name and argument list alone in a statement, as in the following example. If the function has a return value, it is discarded.

```
wait(12);
```

The second method can be used only with functions that have a return value. Because these functions evaluate to a value (that is, their return value), they are valid C expressions and can be used anywhere a C expression can be used. You've already seen an expression with a return value used as the right side of an assignment statement. Here are some more examples.

In the following example, half_of() is a parameter of a function:

```
printf("Half of %d is %d.", x, half_of(x));
```

First, the function half_of() is called with the value of x, and then printf() is called using the values x and half_of(x).

In this second example, multiple functions are being used in an expression:

```
y = half_of(x) + half_of(z);
```

Although half_of() is used twice, the second call could have been any other function. The following code shows the same statement, but not all on one line:

```
a = half_of(x);
b = half_of(z);
y = a + b;
```

The final two examples show effective ways to use the return values of functions. Here, a function is being used with the if statement:

SPACE FOR HINT

```
if ( half_of(x) > 10 )
{
   /* statements; */        /* these could be any statements! */
}
```

If the return value of the function meets the criteria (in this case, if half_of() returns a value greater than 10), the if statement is true, and its statements are executed. If the returned value doesn't meet the criteria, the if's statements are not executed.

The following example is even better:

```
if ( do_a_process() != OKAY )
{
   /* statements; */        /* do error routine */
}
```

Again, I haven't given the actual statements, nor is do_a_process() a real function; however, this is an important example that checks the return value of a process to see whether it ran all right. If it didn't, the statements take care of any error handling or cleanup. This is commonly used with accessing information in files, comparing values, and allocating memory.

If you try to use a function with a void return type as an expression, the compiler generates an error message.

2.1 Types of Function in C Programming Language

You can to declare a C function and call a function in your C program. There are five types of functions. They are

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions that return multiple values.
5. Functions with no arguments and return values.

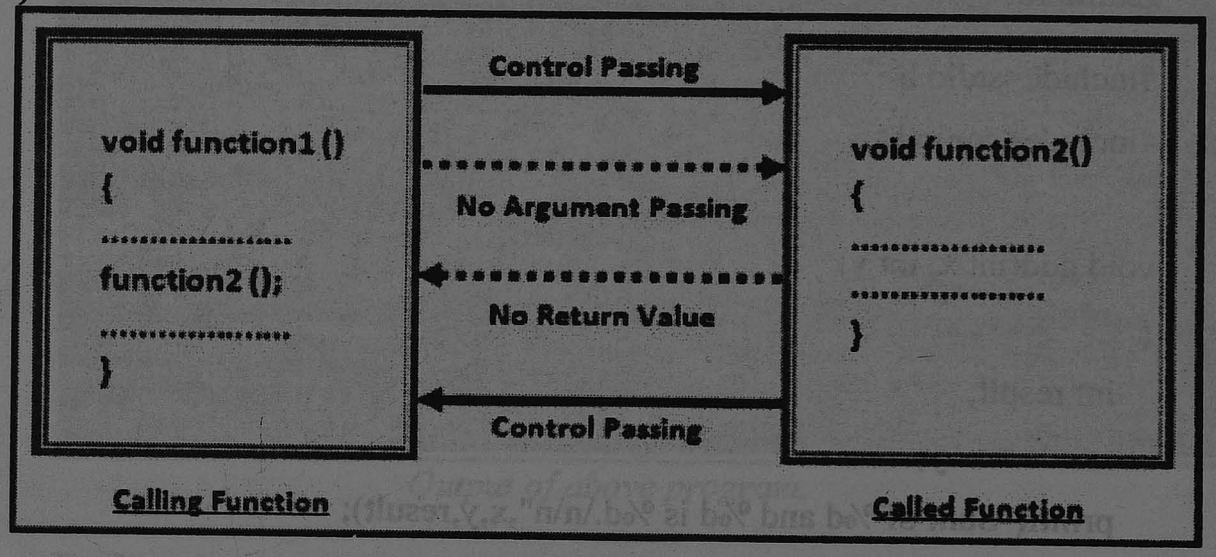
2.1.1. Functions with no arguments and no return value.

A C function without any arguments means you cannot pass data (values like int, char etc) to the called function. Similarly, function with no return type does not pass back data to the **calling function**. It is one of the simplest types of function in C. This type of function which does not return any value cannot be used in an expression it can be used only as independent statement. Let's have an example to illustrate this.

```
#include<stdio.h>
#include<conio.h>
```

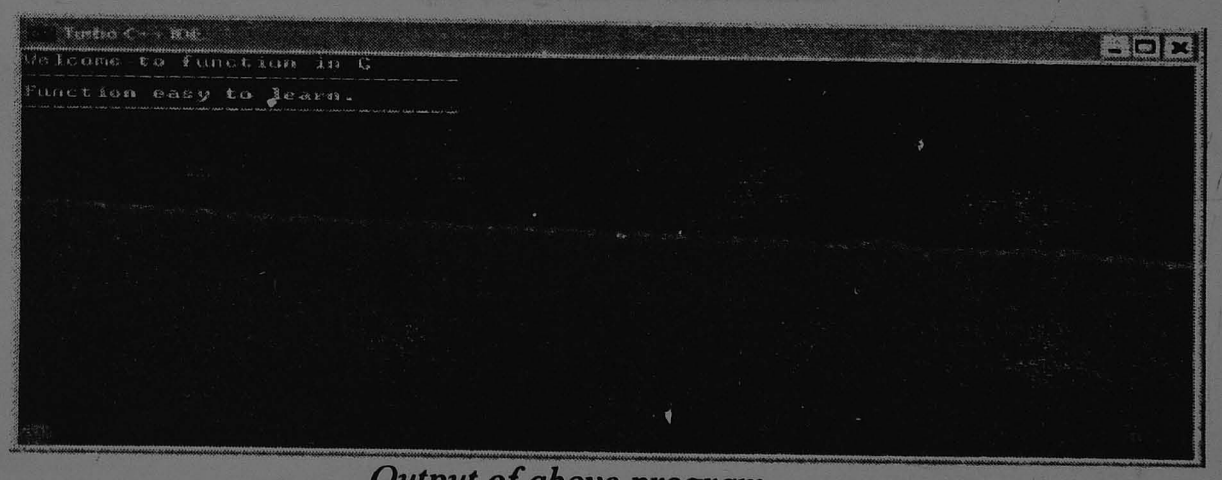
```
void printline()
{
   int i;
   printf("\n");
   for(i=0;i<30;i++)
   {
      printf("-");
   }
   printf("\n");
}

void main()
{
   clrscr();
   printf("Welcome to function in C");
   printline();
   printf("Function easy to learn.");
   printline();
   getch();
}
```

SPACE FOR HINT



Logic of the functions with no arguments and no return value.

Output



Output of above program.

SPACE FOR HINT

Explanation:

The above C program example illustrates that how to declare a function with no argument and no return type.

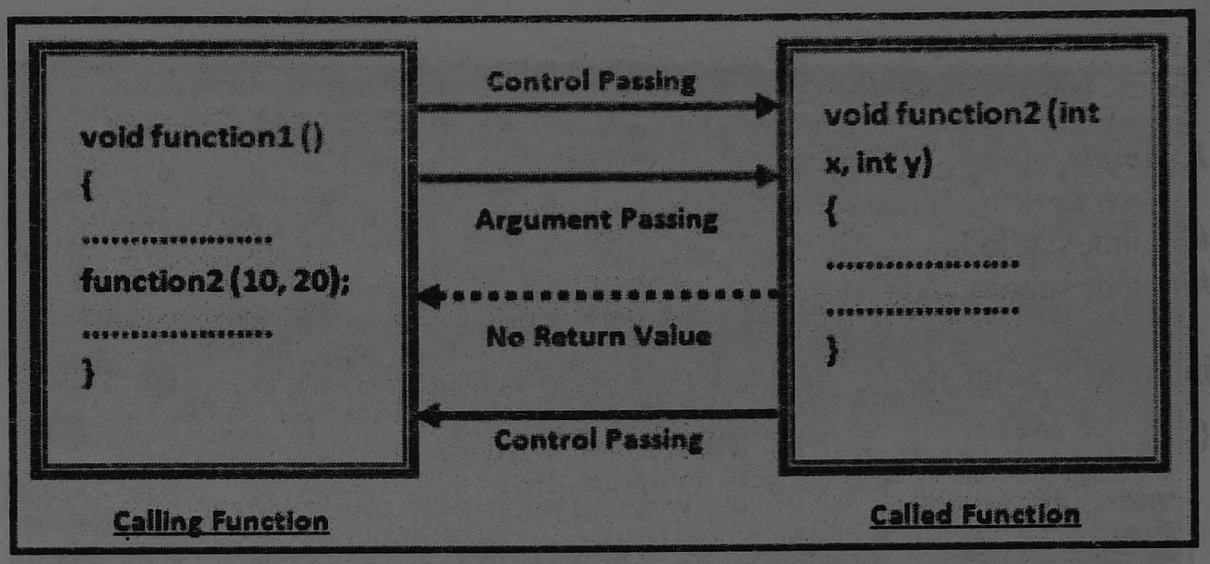
2.1.2 Functions with arguments and no return value.

In the above example what we have noticed that "main()" function has no control over the UDF "printfline()", it cannot control its output. Whenever "main()" calls "printline()", it simply prints line every time. So the result remains the same.

A C function with arguments can perform much better than previous function type. This type of function can accept data from calling function. In other words, you send data to the called function from calling function but you cannot send result data back to the calling function. Rather, it displays the result on the terminal. But we can control the output of function by providing various values as arguments. Let's have an example to get it better.

Example

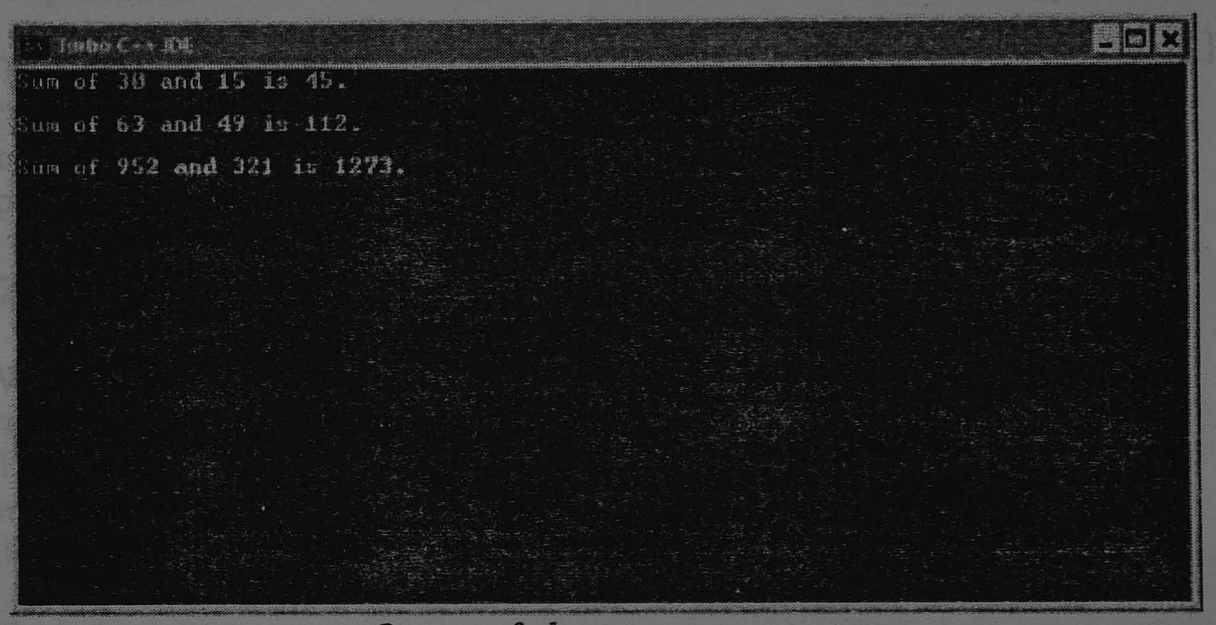
```
#include<stdio.h>
#include<conio.h>

void add(int x, int y)
{
   int result;
   result = x+y;
   printf("Sum of %d and %d is %d.\n\n",x,y,result);
}

void main()
{
   clrscr();
   add(30,15);
   add(63,49);
   add(952,321);
   getch();
}
```



Logic of the function with arguments and no return value.

SPACE FOR HINT

Output



Output of above program.

Explanation:

This program simply sends two integer arguments to the UDF "add()" which, further, calculates its sum and stores in another variable and then prints that value. So simple program to understand.

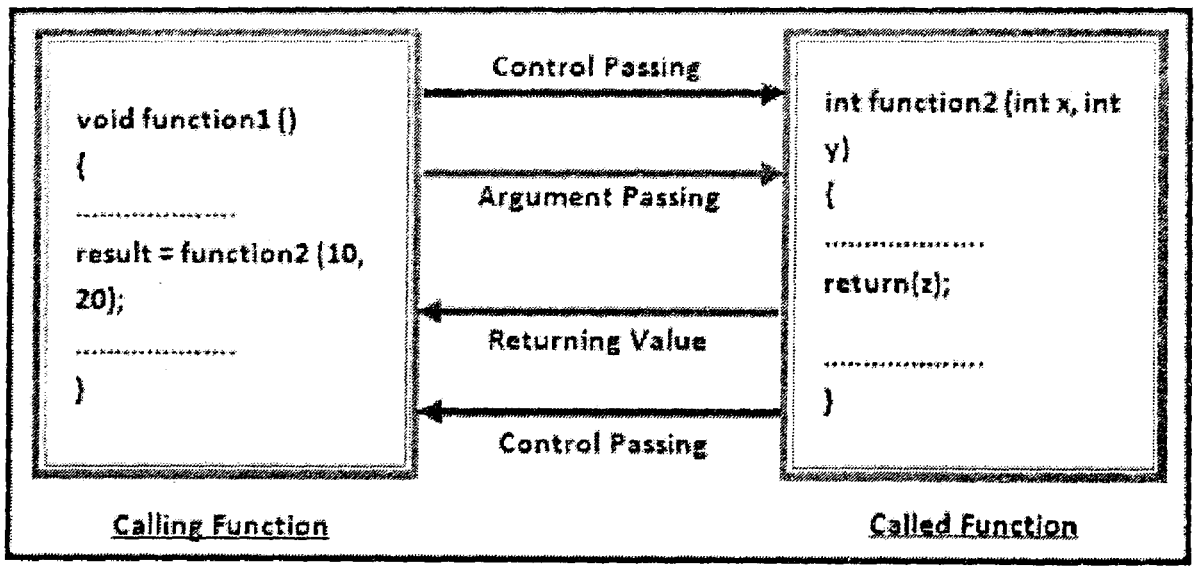
2.1.3 Functions with arguments and return value.

This type of function can send arguments (data) from the calling function to the called function and wait for the result to be returned back from the called function back to the calling function. And this type of function is mostly used in programming world because it can do two way communications; it can accept data as arguments as well as can send back data as return value.

SPACE FOR HINT

Example

```
#include<stdio.h>
#include<conio.h>

int add(int x, int y)
{
   int result;
   result = x+y;
   return(result);
}
void main()
{
   int z;
   clrscr();

   z = add(952,321);
   printf("Result %d.\n\n",add(30,55));
   printf("Result %d.\n\n",z);

   getch();

}
```



Logic of the function with arguments and return value.

Output of the above program.

SPACE FOR HINT

Explanation

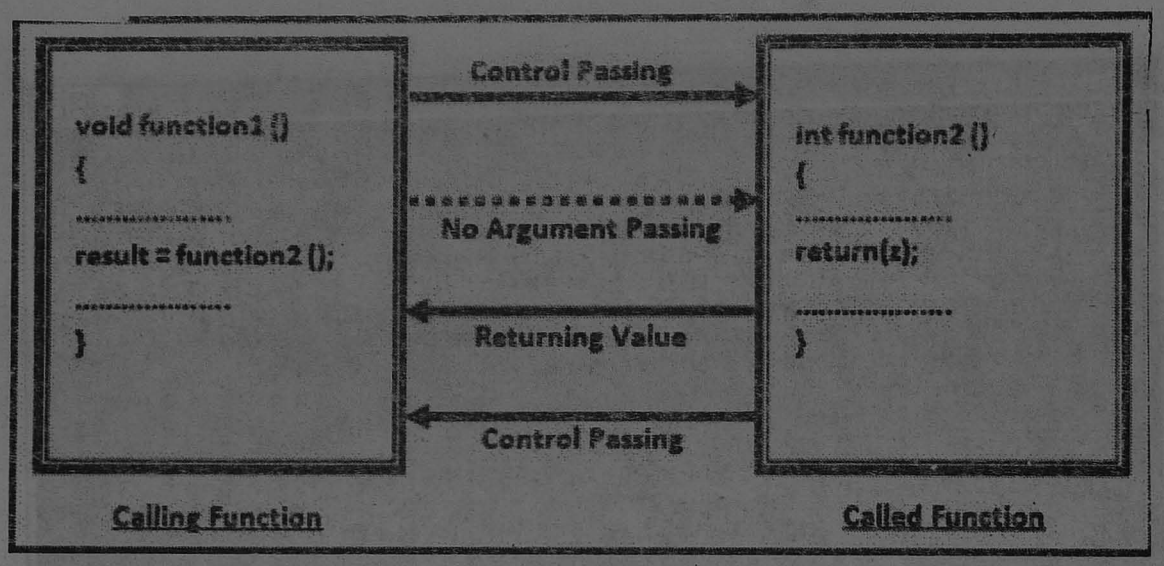
This program sends two integer values (x and y) to the UDF "add()", "add()" function adds these two values and sends back the result to the calling function (in this program to "main()" function). Later result is printed on the terminal.

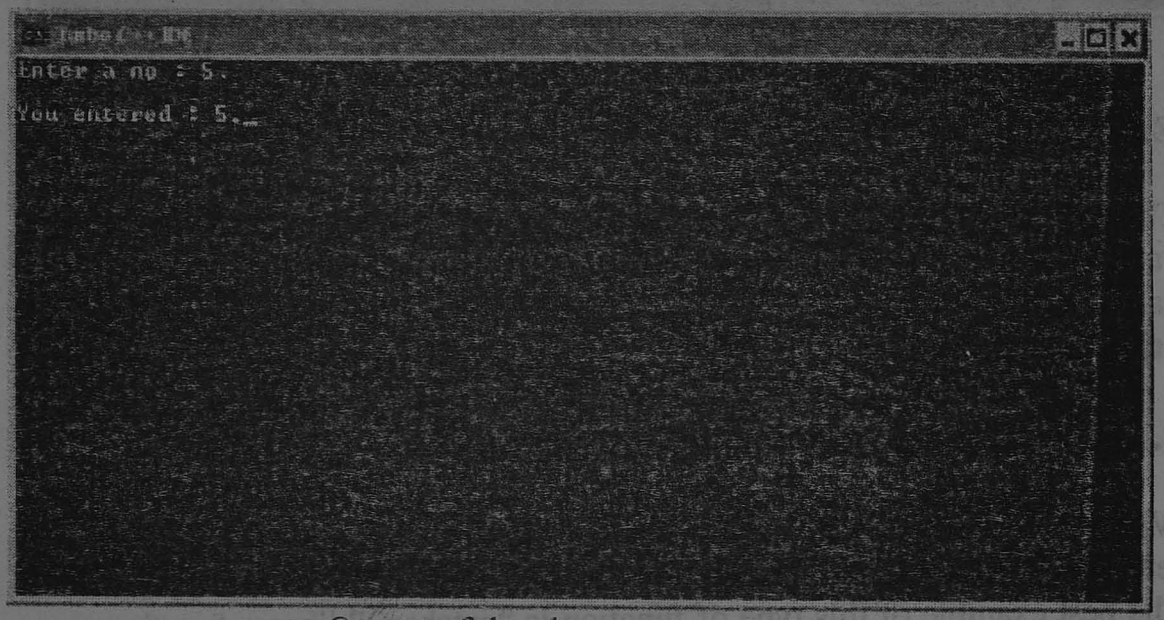
2.1.4 Functions with no arguments but returns value.

We may need a function which does not take any argument but only returns values to the calling function then this type of function is useful. The best example of this type of function is "getchar()" library function which is declared in the header file "stdio.h". We can declare a similar library function of own. Take a look.

```
#include<stdio.h>
#include<conio.h>
 int send()
{
   int no1;
   printf("Enter a no : ");
   scanf("%d",&no1);
   return(no1);
}

void main()
{
   int z;
   clrscr();
   z = send();
   printf("\nYou entered : %d.", z);
   getch();

}
```

SPACE FOR HINT





Output of the above program.

Explanation:

In this program we have a UDF which takes one integer as input from keyboard and sends back to the calling function.

2.1.5 Functions that return multiple values.

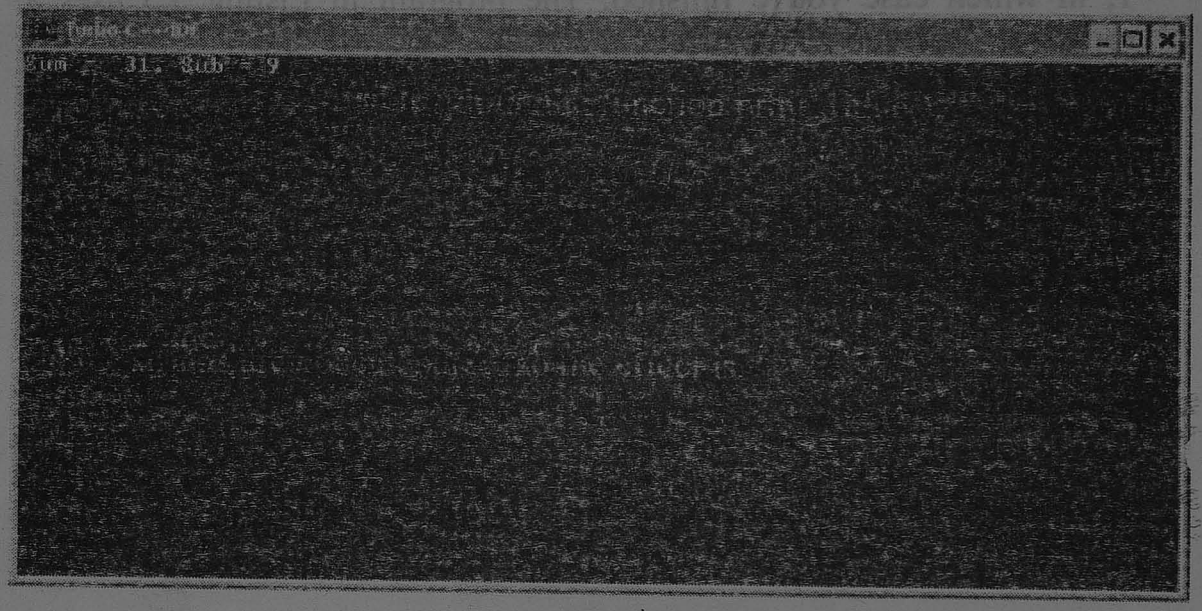
So far, we have learned and seen that in a function, return statement was able to return only single value. That is because; a return statement can return only one value. But if we want to send back more than one value then how we could do this?

We have used arguments to send values to the called function, in the same way we can also use arguments to send back information to the calling function. The arguments that are used to send back data are called **Output Parameters**.

It is a bit difficult for novice because this type of function uses pointer. Let's see an example:

```
#include<stdio.h>
#include<conio.h>

void calc(int x, int y, int *add, int *sub)
{
   *add = x+y;
   *sub = x-y;
}

void main()
{
   int a=20, b=11, p,q;
   clrscr();
   calc(a,b,&p,&q);
   printf("Sum = %d, Sub = %d",p,q);
   getch();

}
```

SPACE FOR HINT



Output of the above program.

Explanation:

Logic of this program is that we call UDF "calc()" and sends argument then it adds and subtract that two values and store that values in their respective pointers. The "*" is known as indirection operator whereas "&" known as address operator. We can get memory address or any variable by simply placing "&" before variable name. In the same way we get value stored at specific memory location by using "*" just before memory address.

2.2 Recursion

The term recursion refers to a situation in which a function calls itself either directly or indirectly. Indirect recursion occurs when one function

SPACE FOR HINT

calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations.

For example, recursion can be used to calculate the factorial of a number. The factorial of a umber x is written x! and is calculated as follows:

x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1

However, you can also calculate x! like this:

x! = x * (x-1)!

Going one step further, you can calculate (x-1)! using the same procedure:

(x-1)! = (x-1) * (x-2)!

You can continue calculating recursively until you're down to a value of 1, in which case you're finished. The program in Listing 5.5 uses a recursive function to calculate factorials. Because the program uses unsigned integers, it's limited to an input value of 8; the factorial of 9 and larger values are outside the allowed range for integers.

using a recursive function to calculate factorials.

```
1:  /* Demonstrates function recursion. Calculates the */
2:  /* factorial of a number. */
3:
4:  #include <stdio.h>
5:
6:  unsigned int f, x;
7:  unsigned int factorial(unsigned int a);
8:
9:  main()
10: {
11:    puts("Enter an integer value between 1 and 8: ");
12:    scanf("%d", &x);
13:
14:    if( x > 8 || x < 1)
15:    {
16:       printf("Only values from 1 to 8 are acceptable!");
17:    }
18:    else
19:    {
20:       f = factorial(x);
21:       printf("%u factorial equals %u\n", x, f);
22:    }
23:
24:    return 0;
```

```
25: }
26:
27: unsigned int factorial(unsigned int a)
28: {
29:    if (a == 1)
30:       return 1;
31:    else
32:    {
33:       a *= factorial(a-1);
34:       return a;
35:    }
36: }
```

SPACE FOR HINT

Enter an integer value between 1 and 8:

6

6 factorial equals 720

2.3 Library Functions and Macros

Checking character types. Handling strings. Doing maths.

C provides a repertoire of standard library functions and macros for specialized purposes (and for the advanced user). These may be divided into various categories. For instance

- Character identification ('ctype.h')
- String manipulation ('string.h')
- Mathematical functions ('math.h')

A program generally has to #include special header files in order to use special functions in libraries.

2.3.1 Character Identification

Some or all of the following functions/macros will be available for identifying and classifying single characters.

Assume that 'true' has any non-zero, integer value and that 'false' has the integer value zero. ch stands for some character, or char type variable.

isalpha(ch)

This returns true if ch is alphabetic and false otherwise. Alphabetic means a..z or A..Z.

isupper(ch)

SPACE FOR HINT

Returns true if the character was upper case. If ch was not an alphabetic character, this returns false.

islower(ch)

Returns true if the character was lower case. If ch was not an alphabetic character, this returns false.

isdigit(ch)

Returns true if the character was a digit in the range 0..9.

isxdigit(ch)

Returns true if the character was a valid hexadecimal digit: that is, a number from 0..9 or a letter a..f or A..F.

isspace(ch)

Returns true if the character was a white space character, that is: a space, a TAB character or a newline.

ispunct(ch)

Returns true if ch is a punctuation character.

isalnum(ch)

Returns true if a character is alphanumeric: that is, alphabeticor digit.

isprint(ch)

Returns true if the character is printable: that is, the character is not a control character.

isgraph(ch)

Returns true if the character is graphic. i.e. if the character is printable (excluding the space)

iscntrl(ch)

Returns true if the character is a control character. i.e. ASCIIvalues 0 to 31 and 127.

isascii(ch)

Returns true if the character is a valid ASCII character: that is, it has a code in the range 0..127.

SPACE FOR HINT

toupper(ch)

This converts the character ch into its upper case counterpart. This does not affect characters which are already upper case, or characters which do not have a particular case, such as digits.

tolower(ch)

This converts a character into its lower case counterpart. It does not affect characters which are already lower case.

toascii(ch)

This strips off bit 7 of a character so that it is in the range 0..127: that is, a valid ASCII character.

Examples

```
/* Demonstration of character utility functions */
/* prints out all the ASCII characters which give  the value "true" for the
listed character fns */

#include <stdio.h>
#include <ctype.h> /* contains character utilities */
#define ALLCHARS ch = 0; isascii(ch); ch++

main () /* A criminally long main program! */
{
char ch;
printf ("VALID CHARACTERS FROM isalpha()\n\n");
for (ALLCHARS)
{
if (isalpha(ch))
{
printf ("%c ",ch);
}
}
printf ("\n\nVALID CHARACTERS FROM isupper()\n\n");
for (ALLCHARS)
{
if (isupper(ch))
{
printf ("%c ",ch);
}
}
printf ("\n\nVALID CHARACTERS FROM islower()\n\n");
for (ALLCHARS)
{
if (islower(ch))
```

SPACE FOR HINT

```
printf ("%c ",ch);
}
}
printf ("\n\nVALID CHARACTERS FROM isdigit()\n\n");
for (ALLCHARS)
{
if (isdigit(ch))
{
printf ("%c ",ch);
}
}
printf ("\n\nVALID CHARACTERS FROM isxdigit()\n\n");
for (ALLCHARS)
{
if (isxdigit(ch))
{
printf ("%c ",ch);
}
}
printf ("\n\nVALID CHARACTERS FROM ispunct()\n\n");
for (ALLCHARS)
{
if (ispunct(ch))
{
printf ("%c ",ch);
}
}
printf ("\n\nVALID CHARACTERS FROM isalnum()\n\n");
for (ALLCHARS)
{
if (isalnum(ch))
{
printf ("%c ",ch);
}
}
printf ("\n\nVALID CHARACTERS FROM iscsym()\n\n");
for (ALLCHARS)
{
if (iscsym(ch))
{
printf ("%c ",ch);
}
}
}
```

Program Output

VALID CHARACTERS FROM isalpha()

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g
h i j
k l m n o p q r s t u v w x y z

VALID CHARACTERS FROM isupper()
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

VALID CHARACTERS FROM islower()
a b c d e f g h i j k l m n o p q r s t u v w x y z

VALID CHARACTERS FROM isdigit()
0 1 2 3 4 5 6 7 8 9

VALID CHARACTERS FROM isxdigit()
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

VALID CHARACTERS FROM ispunct()
! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~

VALID CHARACTERS FROM isalnum()
0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W
X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

VALID CHARACTERS FROM iscsym()
0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W
X Y Z _ a b c d e f g h i j k l m n o p q r s t u v w x y z

SPACE FOR HINT

2.3.2 String Manipulation

The following functions perform useful functions for string handling,

strcat()

This function "concatenates" two strings: that is, it joins them together into one string.
The effect of:

```
char *new,*this, onto[255];
new = strcat(onto,this);
```

is to join the string this onto the string onto. new is a pointer to the complete string; it is identical to onto. Memory is assumed to have been allocated for the starting strings. The string which is to be copied to must be large enough to accept the new string, tagged onto the end. If it is not then unpredictable effects will result. (In some programs the user might get away without declaring enough space for the "onto" string, but in general the results will be garbage, or even a crashed machine.) To join two static strings together, the following code is required:

SPACE FOR HINT

```
char *s1 = "string one";
char *s2 = "string two";
main ()
{
char buffer[255];
strcat(buffer,s1);
strcat(buffer,s2);
}
```

buffer would then contain "string onestring two".

strlen()

This function returns a type int value, which gives the length or number of characters in a string, not including the NULL byte end marker. An example is:

```
int len;
char *string;
len = strlen (string);
```

strcpy()

This function copies a string from one place to another. Use this function in preference to custom routines: it is set up to handle any peculiarities in the way data are stored. An example is

```
char *to,*from;
to = strcpy (to,from);
```

Where to is a pointer to the place to which the string is to be copied and from is the place where the string is to be copied from.

strcmp()

This function compares two strings and returns a value which indicates how they compared. An example:

```
int value;
char *s1,*s2;
value = strcmp(s1,s2);
```

SPACE FOR HINT

The value returned is 0 if the two strings were identical. If the strings were not the same, this function indicates the (ASCII) alphabetical order of the two. s1 > s2, alphabetically, then the
value is '> 0'. If s1 < s2 then the value is < 0. Note that numbers come before letters in the ASCII code sequence and also that upper case comes before lower case.

There are also variations on the theme of the functions above which begin with 'strn' instead of 'str'. These enable the programmer to perform the same actions with the first n characters of a string:

strncat()

This function concatenates two strings by copying the first n characters of this to the end of the onto string.

```
char *onto,*new,*this;
new = strncat(onto,this,n);
```

strncpy()

This function copies the first n characters of a string from one place to another

```
char *to,*from;
int n;
to = strncpy (to,from,n);
```

strncmp()

This function compares the first n characters of two strings

```
int value;
char *s1,*s2;
value = strcmp(s1,s2,n);
```

The following functions perform conversions between strings and floating point/integer types, without needing to use sscanf(). They take a preinitialized string and work out the value represented by that string.

atof()

ASCII to floating point conversion.

```
double x;
char *stringptr;
x = atof(stringptr);
```

atoi()

SPACE FOR HINT

ASCII to integer conversion.

```
int i;
char *stringptr;
i = atoi(stringptr);
```

atol()

ASCII to long integer conversion.

```
long i;
char *stringptr;
i = atol(stringptr);
```

Examples

```
/* String comparison */
#include <stdio.h>
#define TRUE 1
#define MAXLEN 30

main ()
{
char string1[MAXLEN],string2[MAXLEN];
int result;
while (TRUE)
{
printf ("Type in string 1:\n\n");
scanf ("%30s",string1);
printf ("Type in string 2:\n\n");
scanf ("%30s",string2);
result = strcmp (string1,string2);
if (result == 0)
{
printf ("Those strings were the same!\n");
}
if (result > 0)
{
printf ("string1 > string2\n");
}
if (result < 0)
{
printf ("string1 < string 2\n");
}
}
}
```

2.3.3 Mathematical Functions

SPACE FOR HINT

C has a library of standard mathematical functions which can be accessed by #including the appropriate header files ('math.h' etc.). It should be noted that all of these functions work with double or long float type variables.

All of C's mathematical capabilities are written for long variable types. Here is a list of the functions which can be expected in the standard library file. The variables used are all to be declared long

```
int i; /* long int */
double x,y,result; /* long float */
```

The functions themselves must be declared long float or double (which might be done automatically in the mathematics library file, or in a separate file) and any constants must be written in floating point form: for instance, write '7.0' instead of just '7'.

ABS()

MACRO. Returns the unsigned value of the value in parentheses.

fabs()

Find the absolute or unsigned value of the value in parentheses:

```
result = fabs(x);
```

ceil()

Find out what the ceiling integer is: that is, the integer which is just above the value in parentheses. This is like rounding up.

```
i = ceil(x);
/* ceil (2.2) is 3 */
```

floor()

Find out what the floor integer is: that is, the integer which is just below the floating point value in parentheses

```
i = floor(x);
/* floor(2.2) is 2 */
```

exp()

Find the exponential value.

```
result = exp(x);
```

SPACE FOR HINT

result = exp(2.7);

log()

Find the natural (Naperian) logarithm. The value used in the parentheses must be unsigned: that is, it must be greater than zero. It does not have to be declared specifically as unsigned.

e.g.
result = log(x);
result = log(2.71828);

log10()

Find the base 10 logarithm. The value used in the parentheses must be unsigned: that is, it must be greater than zero. It does not have to be declared specifically as unsigned.

result = log10(x);
result = log10(10000);

pow()

Raise a number to the power.

result = pow(x,y); /*raise x to the power y */
result = pow(x,2); /*find x-squared */
result = pow(2.0,3.2); /* find 2 to the power 3.2 ...*/

sqrt()

Find the square root of a number.

result = sqrt(x);
result = sqrt(2.0);

sin()

Find the sine of the angle in radians.

result = sin(x);
result = sin(3.14);

cos()

Find the cosine of the angle in radians.

result = cos(x);
result = cos(3.14);

SPACE FOR HINT

tan()

Find the tangent of the angle in radians.

result = tan(x);
result = tan(3.14);

asin()

Find the arcsine or inverse sine of the value which must lie be-

tween +1.0 and -1.0.
result = asin(x);
result = asin(1.0);

acos()

Find the arccosine or inverse cosine of the value which must lie between +1.0 and -1.0.

result = acos(x);
result = acos(1.0);

atan()

Find the arctangent or inverse tangent of the value.

result = atan(x);
result = atan(200.0);

atan2()

This is a special inverse tangent function for calculating the inverse tangent of x divided by y. This function is set up to find this result more accurately than atan().

result = atan2(x,y);
result = atan2(x/3.14);

sinh()

Find the hyperbolic sine of the value. (Pronounced "shine" or "sinch")

result = sinh(x);
result = sinh(5.0);

cosh()

Find the hyperbolic cosine of the value.

SPACE FOR HINT

```
result = cosh(x);
result = cosh(5.0);
```

tanh()

Find the hyperbolic tangent of the value.

```
result = tanh(x);
result = tanh(5.0);
```

Example

```
/* Maths functions demo #1 */
/* use sin(x) to work out an animated model */
#include <stdio.h>
#include <math.h>
#include <limits.h>
#define TRUE 1
#define AMPLITUDE 30
#define INC 0.02
double pi; /* this may already be defined */
/* in the math file */
main () /* The simple pendulum program */
{ pi = asin(1.0)*2; /* if PI is not defined */
printf ("\nTHE SIMPLE PENDULUM:\n\n\n");
Pendulum();
}
/*************************************************/
Pendulum ()
{ double x, twopi = pi * 2;
int i,position;
while (true)
{
for (x = 0; x < twopi; x += INC)
{
position = (int)(AMPLITUDE * sin(x));
for (i = -AMPLITUDE; i <= AMPLITUDE; i++)
{
if (i == position)
{
putchar('*');
}
else
{
putchar(' ');
}
}
startofline();
}
```

```
}
}

startofline()
{
Maths Errors 209
putchar('\r');
}
```

SPACE FOR HINT

2.4 The C Preprocessor

C provides certain language facilities by means of a preprocessor, which is conceptionally a separate first step in compilation. The two most frequently used features are #include, to include the contents of a file during compilation, and #define, to replace a token by an arbitrary sequence of characters. Other features described in this section include conditional compilation and macros with arguments.

File Inclusion

File inclusion makes it easy to handle collections of #defines and declarations (among other things). Any source line of the form

```
  #include "filename"
or
  #include <filename>
```

is replaced by the contents of the file filename. If the filename is quoted, searching for the file typically begins where the source program was found; if it is not found there, or if the name is enclosed in < and >, searching follows an implementation-defined rule to find the file. An included file may itself contain #include lines.

There are often several #include lines at the beginning of a source file, to include common #define statements and extern declarations, or to access the function prototype declarations for library functions from headers like <stdio.h>. (Strictly speaking, these need not be files; the details of how headers are accessed are implementation-dependent.)

#include is the preferred way to tie the declarations together for a large program. It guarantees that all the source files will be supplied with the same definitions and variable declarations, and thus eliminates a particularly nasty kind of bug. Naturally, when an included file is changed, all files that depend on it must be recompiled.

Macro Substitution

A definition has the form

```
  #define name replacement text
```

SPACE FOR HINT

It calls for a macro substitution of the simplest kind - subsequent occurrences of the token name will be replaced by the replacement text. The name in a #define has the same form as a variable name; the replacement text is arbitrary. Normally the replacement text is the rest of the line, but a long definition may be continued onto several lines by placing a \ at the end of each line to be continued. The scope of a name defined with #define is from its point of definition to the end of the source file being compiled. A definition may use previous definitions. Substitutions are made only for tokens, and do not take place within quoted strings. For example, if YES is a defined name, there would be no substitution in printf("YES") or in YESMAN.

Any name may be defined with any replacement text. For example

```
#define  forever  for (;;)   /* infinite loop */
```

defines a new word, forever, for an infinite loop.

It is also possible to define macros with arguments, so the replacement text can be different for different calls of the macro. As an example, define a macro called max:

```
#define  max(A, B)  ((A) > (B) ? (A) : (B))
```

Although it looks like a function call, a use of max expands into in-line code. Each occurrence of a formal parameter (here A or B) will be replaced by the corresponding actual argument.

Thus the line x = max(p+q, r+s);

will be replaced by the line x = ((p+q) > (r+s) ? (p+q) : (r+s));

So long as the arguments are treated consistently, this macro will serve for any data type; there is no need for different kinds of max for different data types, as there would be with functions.

If you examine the expansion of max, you will notice some pitfalls. The expressions are evaluated twice; this is bad if they involve side effects like increment operators or input and output. For instance

```
max(i++, j++)  /* WRONG */
```

will increment the larger twice. Some care also has to be taken with parentheses to make sure the order of evaluation is preserved; consider what happens when the macro

```
#define square(x)  x * x  /* WRONG */
```

is invoked as square(z+1).

Nonetheless, macros are valuable. One practical example comes from <stdio.h>, in which getchar and putchar are often defined as macros to avoid the run-time overhead of a function call per character processed. The functions in <ctype.h> are also usually implemented as macros.

SPACE FOR HINT

Names may be undefined with #undef, usually to ensure that a routine is really a function, not a macro:

```
#undef getchar
int getchar(void) { ... }
```

Formal parameters are not replaced within quoted strings. If, however, a parameter name is preceded by a # in the replacement text, the combination will be expanded into a quoted string with the parameter replaced by the actual argument. This can be combined with string concatenation to make, for example, a debugging print macro:

```
#define dprint(expr)  printf(#expr " = %g\n", expr)
```

When this is invoked, as in

```
dprint(x/y)
```

the macro is expanded into

```
printf("x/y" " = &g\n", x/y);
```

and the strings are concatenated, so the effect is

```
printf("x/y = &g\n", x/y);
```

Within the actual argument, each " is replaced by \" and each \ by \\, so the result is a legal string constant.

The preprocessor operator ## provides a way to concatenate actual arguments during macro expansion. If a parameter in the replacement text is adjacent to a ##, the parameter is replaced by the actual argument, the ## and surrounding white space are removed, and the result is re-scanned. For example, the macro paste concatenates its two arguments:

#define paste(front, back) front ## back so paste(name, 1) creates the token name1.

Conditional Inclusion

It is possible to control preprocessing itself with conditional statements that are evaluated during preprocessing. This provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

SPACE FOR HINT

The #if line evaluates a constant integer expression (which may not include sizeof, casts, or enum constants). If the expression is non-zerjo, subsequent lines until an #endif or #elif or #else are included. (The preprocessor statement #elif is like else-if.) The expression defined(name) in a #if is 1 if the name has been defined, and 0 otherwise.

For example, to make sure that the contents of a file hdr.h are included only once, the contents of the file are surrounded with a conditional like this:

```
#if !defined(HDR)
#define HDR
/* contents of hdr.h go here */
#endif
```

The first inclusion of hdr.h defines the name HDR; subsequent inclusions will find the name defined and skip down to the #endif. A similar style can be used to avoid including files multiple times. If this style is used consistently, then each header can itself include any other headers on which it depends, without the user of the header having to deal with the interdependence.

This sequence tests the name SYSTEM to decide which version of a header to include:

```
#if SYSTEM == SYSV
   #define HDR "sysv.h"
#elif SYSTEM == BSD
   #define HDR "bsd.h"
#elif SYSTEM == MSDOS
   #define HDR "msdos.h"
#else
   #define HDR "default.h"
#endif
#include HDR
```

The #ifdef and #ifndef lines are specialized forms that test whether a name is defined. The first example of #if above could have been written

```
#ifndef HDR
 #define HDR
 /* contents of hdr.h go here */
 #endif
```

2.5 The Storage Class

What Is Scope?

SPACE FOR HINT

The scope of a variable refers to the extent to which different parts of a program have access to the variable--in other words, where the variable is visible. When referring to C variables, the terms accessibility and visibility are used interchangeably. When speaking about scope, the term variable refers to all C data types: simple variables, arrays, structures, pointers, and so forth. It also refers to symbolic constants defined with the const keyword.

Scope also affects a variable's lifetime: how long the variable persists in memory, or when the variable's storage is allocated and deallocated. First, this chapter examines visibility.

A Demonstration of Scope

Look at the below program. It defines the variable x in line 5, uses printf() to display the value of x in line 11, and then calls the function print_value() to display the value of x again. Note that the function print_value() is not passed the value of x as an argument; it simply uses x as an argument to printf() in line 19.

The variable x is accessible within the function print_value().

```
1: /* Illustrates variable scope. */
2:
3: #include <stdio.h>
4:
5: int x = 999;
6:
7: void print_value(void);
8:
9: main()
10: {
11:    printf("%d\n", x);
12:    print_value();
13:
14:    return 0;
15: }
16:
17: void print_value(void)
18: {
19:    printf("%d\n", x);
20: }
999
999
```

This program compiles and runs with no problems. Now make a minor modification in the program, moving the definition of the variable x to a location within the main() function. The new source code is shown below.

SPACE FOR HINT

The variable x is not accessible within the function print_value().

```
1:  /* Illustrates variable scope. */
2:
3:  #include <stdio.h>
4:
5:  void print_value(void);
6:
7:  main()
8:  {
9:     int x = 999;
10:
11:    printf("%d\n", x);
12:    print_value();
13:
14:    return 0;
15: }
16:
17: void print_value(void)
18: {
19:    printf("%d\n", x);
20: }
```

If you try to compile the above progeram, the compiler generates an error message similar to the following:

```
list1202.c(19) : Error: undefined identifier `x'.
```

Remember that in an error message, the number in parentheses refers to the program line where the error was found. Line 19 is the call to printf() within the print_value() function.

This error message tells you that within the print_value() function, the variable x is undefined or, in other words, not visible. Note, however, that the call to printf() in line 11 doesn't generate an error message; in this part of the program, the variable x is visible.

The only difference between these above two program is where variable x is defined. By moving the definition of x, you change its scope. In the first program, x is an external variable, and its scope is the entire program. It is accessible within both the main() function and the print_value() function. In the second program, x is a local variable, and its scope is limited to within the main() function. As far as print_value() is concerned, x doesn't exist.

Storage Class - revisited

There are four specifiers and two modifiers that can be used to indicate the duration of a variable. These specifiers and modifiers are introduced in the following sections.

SPACE FOR HINT

The auto Specifier
The static Specifier
The extern Specifier
The register Specifier

The auto Specifier

The auto specifier indicates that the memory location of a variable is temporary. In other words, a variable's reserved space in the memory can be erased or relocated when the variable is out of its scope.

Only variables with block scope can be declared with the auto specifier. The auto keyword is rarely used, however, because the duration of a variable with block scope is temporary by default.

The static Specifier

The static specifier, on the other hand, can be applied to variables with either block scope or program scope. When a variable within a function is declared with the static specifier, the variable has a permanent duration. In other words, the memory storage allocated for the variable is not destroyed when the scope of the variable is exited, the value of the variable is maintained outside the scope, and if execution ever returns to the scope of the variable, the last value stored in the variable is still there.

For instance, in the following code portion:

```
int main()
{
  int i;       /* block scope and temporary duration */
  static int j; /* block scope and permanent duration */
  .
  .
  return 0;
}
```

the integer variable i has temporary duration by default. But the other integer variable, j, has permanent duration due to the storage class specifier static.

The following program shows the effect of the static specifier on variables.

Example : Using the static specifier.

SPACE FOR HINT

```
1:  /* Using the static specifier */
2:  #include <stdio.h>
3:  /* the add_two function */
4:  int add_two(int x, int y)
5:  {
6:     static int counter = 1;
7:
8:     printf("This is the function call of %d,\n", counter++);
9:     return (x + y);
10: }
11: /* the main function */
12: main()
13: {
14:    int i, j;
15:
16:    for (i=0, j=5; i<5; i++, j--)
17:       printf("the addition of %d and %d is %d.\n\n",
18:              i, j, add_two(i, j));
19:    return 0;
20: }
```

OUTPUT

```
This is the function call of 1,
the addition of 0 and 5 is 5.
This is the function call of 2,
the addition of 1 and 4 is 5.
This is the function call of 3,
the addition of 2 and 3 is 5.
This is the function call of 4,
the addition of 3 and 2 is 5.
This is the function call of 5,
the addition of 4 and 1 is 5.
```

From the output, you can see that the value saved by counter is indeed incremented by one each time the add_two() function is called, and is retained after the function exits because the integer variable counter is declared with static. Note that counter is only initialized once when the add_two() function is called for the first time.s

The register Specifier

The word register is borrowed from the world of computer hardware. Each computer has a certain number of registers to hold data and perform arithmetic or logical calculations. Because registers are located within the CPU (central processing unit) chip, it's much quicker to access a register than a memory location. Therefore, storing variables in registers may help to speed up your program.

SPACE FOR HINT

The C language provides you with the register specifier. You can apply this specifier to variables when you think it's necessary to put the variables into the computer registers.

However, the register specifier only gives the compiler a suggestion. In other words, a variable specified by the register keyword is not guaranteed to be stored in a register. The compiler can ignore the suggestion if there is no register available, or if some other restrictions have to apply.

It's illegal to take the address of a variable that is declared with the register specifier because the variable is intended to be stored in a register, not in the memory.

In the following portion of code, the integer variable i is declared with the register specifier:

```
int main()
{
  /* block scope with the register specifier */
  register int i;
  . . .
  for (i=0; i<MAX_NUM; i++){
    /* some statements */
  }
  . . .
  return 0;
}
```

The declaration of i suggests that the compiler store the variable in a register. Because i is intensively used in the for loop, storing i in a register may increase the speed of the code shown here.

The extern Specifier

A variable with program scope is also called a global variable.

Here is a question: How can a global variable declared in file A, for instance, be seen in file B? In other words, how does the compiler know that the variable used in file B is actually the same variable declared in file A?

The answer is: Use the extern specifier provided by the C language to allude to a global variable defined elsewhere. In this case, we declare a global variable in file A, and then declare the variable again using the extern specifier in file B.

For instance, suppose you have two global int variables, y and z, that are defined in one file, and then, in another file, you may have the following declarations:

SPACE FOR HINT

```
int x = 0;        /* a global variable */
extern int y;     /* an allusion to a global variable y */
int main()
{
  extern int z;  /* an allusion to a global variable z */
  int i;         /* a local variable */
  .
  .
  .
  return 0;
}
```

As you can see, there are two integer variables, y and z, that are declared with the extern specifier, outside and inside the main() function, respectively. When the compiler sees the two declarations, it knows that the declarations are actually allusions to the global variables y and z that are defined elsewhere.

Scope and Lifetime of Variables

Storage class	Where declared	Visibility(Active)	Lifetime(Alive)
None	Before all functions in a file(may be initialized)	Entire file plus other files where variable s declared with extern	Entire program (Global)
extern	Before all function in a class (cannot be initialized) Extern and the file where originally declared as global	Entire file plus other files where variable is declared	Global
static	Before all functions in a file	Only in that file	Global
None or autc	Inside a function (or a block)	Only in that function or block	Until end of function of block.
register	Inside a function or a block	Only in that function or block	Until end of function of block.
static	Inside a function.	Only in that function	Global

2.6 Multifile Program

SPACE FOR HINT

We are assuming that all the function (including the **main**) are defined in one file. Howe4ver, n real-life programming environment, we may use more than one source files which may be completed separately and linked later to form an executable object code. This approach is very useful because any change n one file does not affect other files thus eliminating the need for recompilation of the entire program.

Multiple source files can share a variable provided it is declared as an external variable appropriately. Variables that are shared by two or more files are global variables and therefore we must declare them accordingly n one file and then explicitly define them with extern in other files. The following program illustrates the use of extern declarations in a multifile program.

The function main in **file1** can reference the variable **m** that is declared as global in file2. Remember, **function1** cannot access the variable **m.** If, however, the **extern int m;** statement is placed before **main,** then both the function could refer to **m**. This can also be achieved by using **extern int m;** statement inside each function n **file1.**

file1.c

```
main()
{
        extern int m ;
        int i;
        .....
        .....
}
function1()
{
        int j;
        .....
        .....
}
```

file2.c

```
int m;  /* global variable */
 function2()
{
        int i;
        .....
        .....
}
function3()
{
        int count;
```

SPACE FOR HINT

```
        .....
        .....
}
```

The extern specifier tells the compiler that the following variable types and names have already been declared elsewhere and no need to create storage space for them. it s the responsibility of the linker to resolve the reference problem . It s important to note that a multifile global variable should be declared without extern in one (and only one) of the files. The extern declaration is done in places where secondary references are made. If we declare a variable as global in two different files used by a single program, then the linker will have a conflict as to which variable to use and, therefore, issues a warning.

The above multifile program can be modified as shown below

file1.c

```
int m;  /*global varable */

main()
{
        int i;
        .....
        .....
}
function1()
{
        int j;
        .....
        .....
}
```

file2.c

```
extern int m; /* global variable */

function2()
{
        int i;
        .....
        .....
}
function3()
{
        int count;
        .....
        .....
}
```

When a function is defined in one file and accessed in another, the later file must include a function declaration. The declaration identifies the function as an external function whose definition appears elsewhere. We usually place such declaration at the beginning of the file, before all functions. Although all functions assumed to be external, it would be a good practice to explicitly declare such functions with the storage class **extern.**

SPACE FOR HINT

2.7 Bitwise Operations

As you may know, the most basic unit of computer data storage is the bit. There are times when being able to manipulate individual bits in your C program's data is very useful. C has several tools that let you do this.

The C bitwise operators let you manipulate the individual bits of integer variables. Remember, a bit is the smallest possible unit of data storage, and it can have only one of two values: 0 or 1. The bitwise operators can be used only with integer types: char, int, and long.

The Shift Operators

Two shift operators shift the bits in an integer variable by a specified number of positions. The << operator shifts bits to the left, and the >> operator shifts bits to the right.

The syntax for these binary operators is

x << n

and

x >> n

Each operator shifts the bits in x by n positions in the specified direction. For a right shift, zeros are placed in the n high-order bits of the variable; for a left shift, zeros are placed in the n low-order bits of the variable. Here are a few examples:

Binary 00001100 (decimal 12) right-shifted by 2 evaluates to binary 00000011 (decimal 3).

Binary 00001100 (decimal 12) left-shifted by 3 evaluates to binary 01100000 (decimal 96).

Binary 00001100 (decimal 12) right-shifted by 3 evaluates to binary 00000001 (decimal 1).

Binary 00110000 (decimal 48) left-shifted by 3 evaluates to binary 10000000 (decimal 128).

SPACE FOR HINT

Under certain circumstances, the shift operators can be used to multiply and divide an integer variable by a power of 2.

Left-shifting an integer by n places has the same effect as multiplying it by 2n, and right-shifting an integer has the same effect as dividing it by 2n. The results of a left-shift multiplication are accurate only if there is no overflow--that is, if no bits are "lost" by being shifted out of the high-order positions. A right-shift division is an integer division, in which any fractional part of the result is lost.

For example, if you right-shift the value 5 (binary 00000101) by one place, intending to divide by 2, the result is 2 (binary 00000010) instead of the correct 2.5, because the fractional part (the .5) is lost. The following program demonstrates the shift operators.

Listing: Using the shift operators.

```
1:  /* Demonstrating the shift operators. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      unsigned int y, x = 255;
8:      int count;
9:
10:     printf("Decimal\t\tshift left by\tresult\n");
11:
12:     for (count = 1; count < 8; count++)
13:     {
14:         y = x << count;
15:         printf("%d\t\t%d\t\t%d\n", x, count, y);
16:     }
17:     printf("\n\nDecimal\t\tshift right by\tresult\n");
18:
19:     for (count = 1; count < 8; count++)
20:     {
21:         y = x >> count;
22:         printf("%d\t\t%d\t\t%d\n", x, count, y);
23:     }
24:     return(0);
25: }
```

Decimal	shift left by	result
255	1	254
255	2	252
255	3	248
255	4	240
255	5	224

SPACE FOR HINT

255	6	192
255	7	128

Decimal	**shift right by**	**result**
255	1	127
255	2	63
255	3	31
255	4	15
255	5	7
255	6	3
255	7	1

The Bitwise Logical Operators

Three bitwise logical operators are used to manipulate individual bits in an integer data type, as shown in the following Table. These operators have names similar to the TRUE/FALSE logical operators you learned about in earlier chapters, but their operations differ.

Table : The bitwise logical operators.

Operator	**Description**
&	AND
|	Inclusive OR
^	Exclusive OR

These are all binary operators, setting bits in the result to 1 or 0 depending on the bits in the operands. They operate as follows:

Bitwise AND sets a bit in the result to 1 only if the corresponding bits in both operands are 1; otherwise, the bit is set to 0. The AND operator is used to turn off, or clear, one or more bits in a value.

Bitwise inclusive OR sets a bit in the result to 0 only if the corresponding bits in both operands are 0; otherwise, the bit is set to 1. The OR operator is used to turn on, or set, one or more bits in a value.

Bitwise exclusive OR sets a bit in the result to 1 if the corresponding bits in the operands are different (if one is 1 and the other is 0); otherwise, the bit is set to 0.

The following are examples of how these operators work:

Operation Example
AND
  11110000
& 01010101

  01010000

SPACE FOR HINT

Inclusive OR

```
 11110000
|01010101
----------
 11110101
```

Exclusive OR

```
  11110000
^ 01010101
----------
  10100101
```

You just read that bitwise AND and bitwise inclusive OR can be used to clear or set, respectively, specified bits in an integer value. Here's what that means. Suppose you have a type char variable, and you want to ensure that the bits in positions 0 and 4 are cleared (that is, equal to 0) and that the other bits stay at their original values. If you AND the variable with a second value that has the binary value 11101110, you'll obtain the desired result. Here's how this works:

In each position where the second value has a 1, the result will have the same value, 0 or 1, as was present in that position in the original variable:

```
0 & 1 == 0
1 & 1 == 1
```

In each position where the second value has a 0, the result will have a 0 regardless of the value that was present in that position in the original variable:

```
0 & 0 == 0
1 & 0 == 0
```

Setting bits with OR works in a similar way. In each position where the second value has a 1, the result will have a 1, and in each position where the second value has a 0, the result will be unchanged:

```
0 | 1 == 1
1 | 1 == 1
0 | 0 == 0
1 | 0 == 1
```

The Complement Operator

The final bitwise operator is the complement operator, ~. This is a unary operator. Its action is to reverse every bit in its operand, changing all 0s

to 1s, and vice versa. For example, ~254 (binary 11111110) evaluates to 1 (binary 00000001).

All the examples in this section have used type char variables containing 8 bits. For larger variables, such as type int and type long, things work exactly the same.

SPACE FOR HINT

C.Y.P

1. **What is Function ?**
2. **What is mean Function Prototype ?**
3. **Explain the various types of Functions in C ?**
4. **Explain about Storage Classes in detail.**
5. **What is Multifile programming**

Table of Contents

UNIT – 3

SPACE FOR HINT

3. INTRODUCTION

What Is an Array?

An array is a collection of data storage locations, each having the same data type and the same name. Each storage location in an array is called an array element. Why do you need arrays in your programs? This question can be answered with an example. If you're keeping track of your business expenses for 1998 and filing your receipts by month, you could have a separate folder for each month's receipts, but it would be more convenient to have a single folder with 12 compartments.

Extend this example to computer programming. Imagine that you're designing a program to keep track of your business expenses. The program could declare 12 separate variables, one for each month's expense total. This approach is analogous to having 12 separate folders for your receipts. Good programming practice, however, would utilize an array with 12 elements, storing each month's total in the corresponding array element. This approach is comparable to filing your receipts in a single folder with 12 compartments. Figure 8.1 illustrates the difference between using individual variables and an array.

Variables are like individual folders, whereas an array is like a single folder with many compartments.

3.1 Defining and processing of array

An array is defined as following :

<type-of-array> <name-of-array> [<number of elements in array>];

type-of-array: It is the type of elements that an array stores. If array stores character elements then type of array is 'char'. If array stores integer elements then type of array is 'int'. Besides these native types, if type of elements in array is structure objects then type of array becomes the structure.

name-of-array: This is the name that is given to array. It can be any string but it is usually suggested that some can of standard should be followed while naming arrays. At least the name should be in context with what is being stored in the array.

[number of elements]: This value in subscripts [] indicates the number of elements the array stores.
For example, an array of five characters can be defined as :

```
char arr[5];
```

SPACE FOR HINT

3.2 Initializing of Array

An array can be initialized in many ways as shown in the code-snippets below.

Initializing each element separately. For example :

```
int arr[10];
int i = 0;
for(i=0;i<sizeof(arr);i++)
{
  arr[i] = i; // Initializing each element seperately
}
```

Initializing array at the time of declaration. For example :

```
int arr[] = {'1','2','3','4','5'};
```

In the above example an array of five integers is declared. Note that since we are initializing at the time of declaration so there is no need to mention any value in the subscripts []. The size will automatically be calculated from the number of values. In this case, the size will be 5.

4. Single-Dimensional Arrays

A single-dimensional array has only a single subscript. A subscript is a number in brackets that follows an array's name. This number can identify the number of individual elements in the array. An example should make this clear. For the business expenses program, you could use the following line to declare an array of type float:

```
float expenses[12];
```

The array is named expenses, and it contains 12 elements. Each of the 12 elements is the exact equivalent of a single float variable. All of C's data types can be used for arrays. C array elements are always numbered starting at 0, so the 12 elements of expenses are numbered 0 through 11. In the preceding example, January's expense total would be tored in expenses[0], February's in expenses[1], and so on.

When you declare an array, the compiler sets aside a block of memory large enough to hold the entire array. Individual array elements are stored in sequential memory locations.

The location of array declarations in your source code is important. An array element can be used in your program anywhere a nonarray variable of the same type can be used. Individual elements of the array are accessed by using the array name followed by the element subscript enclosed in square brackets.

SPACE FOR HINT

For example, the following statement stores the value 89.95 in the second array element (remember, the first array element is expenses[0], not expenses[1]):

```
expenses[1] = 89.95;
```

Likewise, the statement

```
expenses[10] = expenses[11];
```

assigns a copy of the value that is stored in array element expenses[11] into array element expenses[10]. When you refer to an array element, the array subscript can be a literal constant, as in these examples. However, your programs might often use a subscript that is a C integer variable or expression, or even another array element. Here are some examples:

```
float expenses[100];
int a[10];

/* additional statements go here */
expenses[i] = 100;        /* i is an integer variable */
expenses[2 + 3] = 100;    /* equivalent to expenses[5] */
expenses[a[2]] = 100;     /* a[] is an integer array */
```

That last example might need an explanation. If, for instance, you have an integer array named a[] and the value 8 is stored in element a[2], writing

```
expenses[a[2]]
```

has the same effect as writing

```
expenses[8];
```

When you use arrays, keep the element numbering scheme in mind: In an array of n ele-ments, the allowable subscripts range from 0 to n-1. If you use the subscript value n, you might get program errors. The C compiler doesn't recognize whether your program uses an array subscript that is out of bounds. Your program compiles and links, but out-of-range subscripts generally produce erroneous results.

Remember that array elements start with 0, not 1. Also remember that the last element is one less than the number of elements in the array. For example, an array with 10 elements contains elements 0 through 9.

Sometimes you might want to treat an array of n elements as if its elements were numbered 1 through n. For instance, in the previous example, a more natural method might be to store January's expense total in expenses[1], February's in expenses[2], and so on. The simplest

SPACE FOR HINT

way to do this is to declare the array with one more element than needed, and ignore element 0. In this case, you would declare the array as follows. You could also store some related data in element 0 (the yearly expense total, perhaps).

float expenses[13];

The following program demonstrates the use of an array. This is a simple program with no real practical use; it's for demonstration purposes only.

EXPENSES.C demonstrates the use of an array.

```
1:  /* EXPENSES.C - Demonstrates use of an array */
2:
3:  #include <stdio.h>
4:
5:  /* Declare an array to hold expenses, and a counter variable */
6:
7:  float expenses[13];
8:  int count;
9:
10: main()
11: {
12:     /* Input data from keyboard into array */
13:
14:     for (count = 1; count < 13; count++)
15:     {
16:         printf("Enter expenses for month %d: ", count);
17:         scanf("%f", &expenses[count]);
18:     }
19:
20:     /* Print array contents */
21:
22:     for (count = 1; count < 13; count++)
23:     {
24:         printf("Month %d = $%.2f\n", count, expenses[count]);
25:     }
26:     return 0;
27: }
```

```
Enter expenses for month 1: 100
Enter expenses for month 2: 200.12
Enter expenses for month 3: 150.50
Enter expenses for month 4: 300
Enter expenses for month 5: 100.50
Enter expenses for month 6: 34.25
Enter expenses for month 7: 45.75
Enter expenses for month 8: 195.00
Enter expenses for month 9: 123.45
```

```
Enter expenses for month 10: 111.11
Enter expenses for month 11: 222.20
Enter expenses for month 12: 120.00
Month 1 = $100.00
Month 2 = $200.12
Month 3 = $150.50
Month 4 = $300.00
Month 5 = $100.50
Month 6 = $34.25
Month 7 = $45.75
Month 8 = $195.00
Month 9 = $123.45
Month 10 = $111.11
Month 11 = $222.20
Month 12 = $120.00
```

SPACE FOR HINT

5. Multidimensional Arrays

A multidimensional array has more than one subscript. A two-dimensional array has two subscripts, a three-dimensional array has three subscripts, and so on. There is no limit to the number of dimensions a C array can have.

You can declare arrays with as many dimensions as your compiler allows.

The general form of declaring a N-dimensional array is

data-type Array-Name[Array-Size1][Array-Size2]. . .[Array-SizeN];

where N can be any positive integer.

Because the two-dimensional array, which is widely used, is the simplest form of the multidimensional array.

For example, the following statement declares a two-dimensional integer array:

int array_int[2][3];

Here there are two pairs of brackets that represent two dimensions with a size of 2 and 3 integer elements, respectively.

For example, you might write a program that plays checkers. The checkerboard contains 64 squares arranged in eight rows and eight columns. Your program could represent the board as a two-dimensional array, as follows:

int checker[8][8];

SPACE FOR HINT

The resulting array has 64 elements: checker[0][0], checker[0][1], checker[0][2]...checker[7][6], checker[7][7].

A two-dimensional array has a row-and-column structure.

Similarly, a three-dimensional array could be thought of as a cube. Four-dimensional arrays (and higher) are probably best left to your imagination. All arrays, no matter how many dimensions they have, are stored sequentially in memory.

Naming and Declaring Arrays

An array name must be unique. It can't be used for another array or for any other identifier (variable, constant, and so on). As you have probably realized, array declarations follow the same form as declarations of nonarray variables, except that the number of elements in the array must be enclosed in square brackets immediately following the array name.

When you declare an array, you can specify the number of elements with a literal constant or with a symbolic constant created with the #define directive. Thus, the following:

```
#define MONTHS 12
int array[MONTHS];
```

is equivalent to this statement:

```
int array[12];
```

With most compilers, however, you can't declare an array's elements with a symbolic constant created with the const keyword:

```
const int MONTHS = 12;
int array[MONTHS];        /* Wrong! */
```

It is another program demonstrating the use of a single-dimensional array. GRADES.C uses an array to store 10 grades.

GRADES.C stores 10 grades in an array.

```
1:  /*GRADES.C - Sample program with array */
2:  /* Get 10 grades and then average them */
3:
4:  #include <stdio.h>
5:
6:  #define MAX_GRADE 100
7:  #define STUDENTS  10
8:
```

```
10:
11: int idx;
12: int total = 0;          /* used for average */
13:
14: main()
15: {
16:    for( idx=0;idx< STUDENTS;idx++)
17:    {
18:       printf( "Enter Person %d's grade: ", idx +1);
19:       scanf( "%d", &grades[idx] );
20:
21:       while ( grades[idx] > MAX_GRADE )
22:       {
23:          printf( "\nThe highest grade possible is %d",
24                MAX_GRADE );
25:          printf( "\nEnter correct grade: " );
26:          scanf( "%d", &grades[idx] );
27:       }
28:
29:       total += grades[idx];
30:    }
31:
32:    printf( "\n\nThe average score is %d\n", ( total / STUDENTS) );
33:
34:    return (0);
35: }
```

SPACE FOR HINT

OUTPUT

```
Enter Person 1's grade: 95
Enter Person 2's grade: 100
Enter Person 3's grade: 60
Enter Person 4's grade: 105
The highest grade possible is 100
Enter correct grade: 100
Enter Person 5's grade: 25
Enter Person 6's grade: 0
Enter Person 7's grade: 85
Enter Person 8's grade: 85
Enter Person 9's grade: 95
Enter Person 10's grade: 85
The average score is 73
```

Initializing Arrays

You can initialize all or part of an array when you first declare it. Follow the array declaration with an equal sign and a list of values enclosed in braces and separated by commas. The listed values are assigned in order to array elements starting at number 0. For example, the following code

SPACE FOR HINT

assigns the value 100 to array[0], 200 to array[1], 300 to array[2], and 400 to array[3]:

```
int array[4] = { 100, 200, 300, 400 };
```

If you omit the array size, the compiler creates an array just large enough to hold the initialization values. Thus, the following statement would have exactly the same effect as the previous array declaration statement:

```
int array[] = { 100, 200, 300, 400 };
```

You can, however, include too few initialization values, as in this example:

```
int array[10] = { 1, 2, 3 };
```

If you don't explicitly initialize an array element, you can't be sure what value it holds when the program runs. If you include too many initializers (more initializers than array elements), the compiler detects an error.

Initializing Multidimensional Arrays

Multidimensional arrays can also be initialized. The list of initialization values is assigned to array elements in order, with the last array subscript changing first. For example:

```
int array[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

results in the following assignments:

```
array[0][0] is equal to 1
array[0][1] is equal to 2
array[0][2] is equal to 3
array[1][0] is equal to 4
array[1][1] is equal to 5
array[1][2] is equal to 6
...
array[3][1] is equal to 11
array[3][2] is equal to 12
```

When you initialize multidimensional arrays, you can make your source code clearer by using extra braces to group the initialization values and also by spreading them over several lines. The following initialization is equivalent to the one just given:

```
int array[4][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };
```

Remember, initialization values must be separated by a comma--even when there is a brace between them. Also, be sure to use braces in pairs--a closing brace for every opening brace--or the compiler becomes confused.

SPACE FOR HINT

Now look at an example that demonstrates the advantages of arrays. Listing 8.3, RANDOM.C, creates a 1,000-element, three-dimensional array and fills it with random numbers. The program then displays the array elements on-screen. Imagine how many lines of source code you would need to perform the same task with nonarray variables.

You see a new library function, getch(), in this program. The getch() function reads a single character from the keyboard. In the following program getch() pauses the program until the user presses a key.

RANDOM.C creates a multidimensional array.

```
1:  /* RANDOM.C - Demonstrates using a multidimensional array */
2:
3:  #include <stdio.h>
4:  #include <stdlib.h>
5:  /* Declare a three-dimensional array with 1000 elements */
6:
7:  int random_array[10][10][10];
8:  int a, b, c;
9:
10: main()
11: {
12:    /* Fill the array with random numbers. The C library */
13:    /* function rand() returns a random number. Use one */
14:    /* for loop for each array subscript. */
15:
16:    for (a = 0; a < 10; a++)
17:    {
18:       for (b = 0; b < 10; b++)
19:       {
20:          for (c = 0; c < 10; c++)
21:          {
22:             random_array[a][b][c] = rand();
23:          }
24:       }
25:    }
26:
27:    /* Now display the array elements 10 at a time */
28:
29:    for (a = 0; a < 10; a++)
30:    {
31:       for (b = 0; b < 10; b++)
32:       {
33:          for (c = 0; c < 10; c++)
```

SPACE FOR HINT

```
34:          {
35:             printf("\nrandom_array[%d][%d][%d] = ", a, b, c);
36:             printf("%d", random_array[a][b][c]);
37:          }
38:          printf("\nPress Enter to continue, CTRL-C to quit.");
39:
40:          getchar();
41:       }
42:    }
43:    return 0;
44: }       /* end of main() */
```

OUTPUT

```
random_array[0][0][0] = 346
random_array[0][0][1] = 130
random_array[0][0][2] = 10982
random_array[0][0][3] = 1090
random_array[0][0][4] = 11656
random_array[0][0][5] = 7117
random_array[0][0][6] = 17595
random_array[0][0][7] = 6415
random_array[0][0][8] = 22948
random_array[0][0][9] = 31126
Press Enter to continue, CTRL-C to quit.
random_array[0][1][0] = 9004
random_array[0][1][1] = 14558
random_array[0][1][2] = 3571
random_array[0][1][3] = 22879
random_array[0][1][4] = 18492
random_array[0][1][5] = 1360
random_array[0][1][6] = 5412
random_array[0][1][7] = 26721
random_array[0][1][8] = 22463
random_array[0][1][9] = 25047
Press Enter to continue, CTRL-C to quit
...         ...
random_array[9][8][0] = 6287
random_array[9][8][1] = 26957
random_array[9][8][2] = 1530
random_array[9][8][3] = 14171
random_array[9][8][4] = 6951
random_array[9][8][5] = 213
random_array[9][8][6] = 14003
random_array[9][8][7] = 29736
random_array[9][8][8] = 15028
random_array[9][8][9] = 18968
Press Enter to continue, CTRL-C to quit.
random_array[9][9][0] = 28559
random_array[9][9][1] = 5268
```

```
random_array[9][9][2] = 20182
random_array[9][9][3] = 3633
random_array[9][9][4] = 24779
random_array[9][9][5] = 3024
random_array[9][9][6] = 10853
random_array[9][9][7] = 28205
random_array[9][9][8] = 8930
random_array[9][9][9] = 2873
Press Enter to continue, CTRL-C to quit.
```

SPACE FOR HINT

Maximum Array Size

The size of an array in bytes depends on the number of elements it has, as well as each element's size. Element size depends on the data type of the array and your computer. The sizes for each numeric data type, given in the following table for your convenience. These are the data type sizes for many PCs.

Table : Storage space requirements for numeric data types for many PCs.

Element Data Type	Element Size (Bytes)
Int	2 or 4
Short	2
Long	4
Float	4
Double	8

To calculate the storage space required for an array, multiply the number of elements in the array by the element size. For example, a 500-element array of type float requires storage space of 500 * 4 = 2000 bytes.

You can determine storage space within a program by using C's sizeof() operator; sizeof() is a unary operator, not a function.It takes as its argument a variable name or the name of a data type and returns the size, in bytes, of its argument. The use of sizeof() is illustrated in Listing 8.4.

The following program Using the sizeof() operator to determine storage space requirements for an array.

```
1:  /* Demonstrates the sizeof() operator */
2:
3:  #include <stdio.h>
4:
5:  /* Declare several 100-element arrays */
6:
7:  int intarray[100];
8:  float floatarray[100];
9:  double doublearray[100];
```

SPACE FOR HINT

```
10:
11: main()
12: {
13:    /* Display the sizes of numeric data types */
14:
15:    printf("\n\nSize of int = %d bytes", sizeof(int));
16:    printf("\nSize of short = %d bytes", sizeof(short));
17:    printf("\nSize of long = %d bytes", sizeof(long));
18:    printf("\nSize of float = %d bytes", sizeof(float));
19:    printf("\nSize of double = %d bytes", sizeof(double));
20:
21:    /* Display the sizes of the three arrays */
22:
23:    printf("\nSize of intarray = %d bytes", sizeof(intarray));
24:    printf("\nSize of floatarray = %d bytes",
25           sizeof(floatarray));
26:    printf("\nSize of doublearray = %d bytes\n",
27           sizeof(doublearray));
28:
29:    return 0;
30: }
```

The following output is from a 16-bit Windows 3.1 machine:

```
Size of int = 2 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 200 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

You would see the following output on a 32-bit Windows NT machine, as well as a 32-bit UNIX machine:

```
Size of int = 4 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 400 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

Summary

This chapter introduced numeric arrays, a powerful data storage method that lets you group a number of same-type data items under the same group name. Individual items, or elements, in an array are identified

using a subscript after the array name. Computer programming tasks that involve repetitive data processing lend themselves to array storage.

SPACE FOR HINT

Like nonarray variables, arrays must be declared before they can be used. Optionally, array elements can be initialized when the array is declared.

6. Using Strings

Variables of type char can hold only a single character, so they have limited usefulness. You also need a way to store strings, which are sequences of characters. A person's name and address are examples of strings. Although there is no special data type for strings, C handles this type of information with arrays of characters.

6.1 Arrays of Characters

To hold a string of six characters, for example, you need to declare an array of type char with seven elements. Arrays of type char are declared like arrays of other data types. For example, the statement

```
char string[10];
```

declares a 10-element array of type char. This array could be used to hold a string of nine or fewer characters.

"But wait," you might be saying. "It's a 10-element array, so why can it hold only nine characters? In C, a string is defined as a sequence of characters ending with the null character, a special character represented by \0. Although it's represented by two characters (backslash and zero), the null character is interpreted as a single character and has the ASCII value of 0. It's one of C's escape sequences.

When a C program stores the string Alabama, for example, it stores the seven characters A, l, a, b, a, m, and a, followed by the null character \0, for a total of eight characters. Thus, a character array can hold a string of characters numbering one less than the total number of elements in the array.

A type char variable is one byte in size, so the number of bytes in an array of type char variables is the same as the number of elements in the array.

6.2 Initializing Character Arrays

Like other C data types, character arrays can be initialized when they are declared. Character arrays can be assigned values element by element, as shown here:

SPACE FOR HINT

```
char string[10] = { `A', `l', `a', `b', `a', `m', `a', `\0' };
```

It's more convenient, however, to use a literal string, which is a sequence of characters enclosed in double quotes:

```
char string[10] = "Alabama";
```

When you use a literal string in your program, the compiler automatically adds the terminating null character at the end of the string. If you don't specify the number of subscripts when you declare an array, the compiler calculates the size of the array for you. Thus, the following line creates and initializes an eight-element array:

```
char string[] = "Alabama";
```

Remember that strings require a terminating null character. The C functions that manipulate strings determine string length by looking for the null character. These functions have no other way of recognizing the end of the string. If the null character is missing, your program thinks that the string extends until the next null character in memory. Pesky program bugs can result from this sort of error.

Strings and Pointers

You've seen that strings are stored in arrays of type char, with the end of the string (which might not occupy the entire array) marked by the null character. Because the end of the string is marked, all you need in order to define a given string is something that points to its beginning. (Is points the right word? Indeed it is!)

you know that the name of an array is a pointer to the first element of the array. Therefore, for a string that's stored in an array, you need only the array name in order to access it. In fact, using the array's name is C's standard method of accessing strings.

To be more precise, using the array's name to access strings is the method the C library functions expect. The C standard library includes a number of functions that manipulate strings. To pass a string to one of these functions, you pass the array name. The same is true of the string display functions printf() and puts().

6.3 Strings Without Arrays

The string is defined by the character array's name and a null character. The array's name is a type char pointer to the beginning of the string. The null marks the string's end. The actual space occupied by the string in an array is incidental. In fact, the only purpose the array serves is to provide allocated space for the string.

What if you could find some memory storage space without allocating an array? You could then store a string with its terminating null character there instead. A pointer to the first character could serve to specify the string's beginning just as if the string were in an allocated array.

SPACE FOR HINT

How do you go about finding memory storage space? There are two methods: One allocates space for a literal string when the program is compiled, and the other uses the malloc() function to allocate space while the program is executing, a process known as dynamic allocation.

The malloc() Function

The malloc() function is one of C's memory allocation functions. When you call malloc(), you pass it the number of bytes of memory needed. malloc() finds and reserves a block of memory of the required size and returns the address of the first byte in the block. You don't need to worry about where the memory is found; it's handled automatically.

The malloc() function returns an address, and its return type is a pointer to type void. Why void? A pointer to type void is compatible with all data types. Because the memory allocated by malloc() can be used to store any of C's data types, the void return type is appropriate.

```
#include <stdlib.h>
void *malloc(size_t size);
```

malloc() allocates a block of memory that is the number of bytes stated in size. By allocating memory as needed with malloc() instead of all at once when a program starts, you can use a computer's memory more efficiently. When using malloc(), you need to include the STDLIB.H header file. Some compilers have other header files that can be included; for portability, however, it's best to include STDLIB.H.

malloc() returns a pointer to the allocated block of memory. If malloc() was unable to allocate the required amount of memory, it returns null. Whenever you try to allocate memory, you should always check the return value, even if the amount of memory to be allocated is small.

Example 1

```
#include <stdlib.h>
#include <stdio.h>
main()
{
  /* allocate memory for a 100-character string */
  char *str;
  if (( str = (char *) malloc(100)) == NULL)
  {
    printf( "Not enough memory to allocate buffer\n");
```

SPACE FOR HINT

```
    exit(1);
  }
  printf( "String was allocated!"\n );
  return 0;
}
```

Example 2

```
/* allocate memory for an array of 50 integers */
int *numbers;
numbers = (int *) malloc(50 * sizeof(int));
```

Example 3

```
/* allocate memory for an array of 10 float values */
float *numbers;
numbers = (float *) malloc(10 * sizeof(float));
```

Using the malloc() Function

You can use malloc() to allocate memory to store a single type char First, declare a pointer to type char:

```
char *ptr;
```

Next, call malloc() and pass the size of the desired memory block. Because a type char usually occupies one byte, you need a block of one byte. The value returned by malloc() is assigned to the pointer:

```
ptr = malloc(1);
```

This statement allocates a memory block of one byte and assigns its address to ptr. Unlike variables that are declared in the program, this byte of memory has no name. Only the pointer can reference the variable. For example, to store the character `x' there, you would write

```
*ptr = `x';
```

Allocating storage for a string with malloc() is almost identical to using malloc() to allocate space for a single variable of type char. The main difference is that you need to know the amount of space to allocate--the maximum number of characters in the string. This maximum depends on the needs of your program. For this example, say you want to allocate space for a string of 99 characters, plus one for the terminating null character, for a total of 100. First you declare a pointer to type char, and then you call malloc():

```
char *ptr;
ptr = malloc(100);
```

Now ptr points to a reserved block of 100 bytes that can be used for string storage and manipulation. You can use ptr just as though your program had explicitly allocated that space with the following array declaration:

SPACE FOR HINT

```
char ptr[100];
```

Using malloc() lets your program allocate storage space as needed in response to demand. Of course, available space is not unlimited; it depends on the amount of memory installed in your computer and on the program's other storage requirements. If not enough memory is available, malloc() returns 0 (null). Your program should test the return value of malloc() so that you'll know the memory requested was allocated successfully. You should always test malloc()'s return value against the symbolic constant NULL, which is defined in STDLIB.H. Listing 10.3 illustrates the use of malloc(). Any program using malloc() must #include the header file STDLIB.H.

Example: Using the malloc() function to allocate storage space for string data.

```
1: /* Demonstrates the use of malloc() to allocate storage */
2: /* space for string data. */
3:
4: #include <stdio.h>
5: #include <stdlib.h>
6:
7: char count, *ptr, *p;
8:
9: main()
10: {
11:    /* Allocate a block of 35 bytes. Test for success. */
12:    /* The exit() library function terminates the program. */
13:
14:    ptr = malloc(35 * sizeof(char));
15:
16:    if (ptr == NULL)
17:    {
18:       puts("Memory allocation error.");
19:       exit(1);
20:    }
21:
22:    /* Fill the string with values 65 through 90, */
23:    /* which are the ASCII codes for A-Z. */
24:
25:    /* p is a pointer used to step through the string. */
26:    /* You want ptr to remain pointed at the start */
27:    /* of the string. */
28:
```

SPACE FOR HINT

```
29:   p = ptr;
30:
31:   for (count = 65; count < 91 ; count++)
32:      *p++ = count;
33:
34:   /* Add the terminating null character. */
35:
36:   *p = '\0';
37:
38:   /* Display the string on the screen. */
39:
40:   puts(ptr);
41:
42:   return 0;
43: }
```

OUTPUT

ABCDEFGHIJKLMNOPQRSTUVWXYZ

6.4 Displaying Strings and Characters

If your program uses string data, it probably needs to display the data on the screen at some time. String display is usually done with either the puts() function or the printf() function.

The puts() Function

You've seen the puts() library function in some of the programs in this book. Thc puts() function puts a string on-screen--hence its name. A pointer to the string to be displayed is the only argument puts() takes. Because a literal string evaluates as a pointer to a string, puts() can be used to display literal strings as well as string variables. The puts() function
automatically inserts a newline character at the end of each string it displays, so each subsequent string displayed with puts() is on its own line.

The following programillustrates the use of puts().

Using the puts() function to display text on-screen.

```
1:  /* Demonstrates displaying strings with puts(). */
2:
3:  #include <stdio.h>
4:
5:  char *message1 = "C";
6:  char *message2 = "is the";
7:  char *message3 = "best";
8:  char *message4 = "programming";
```

```
9:  char *message5 = "language!!";
10:
11: main()
12: {
13:    puts(message1);
14:    puts(message2);
15:    puts(message3);
16:    puts(message4);
17:    puts(message5);
18:
19:    return 0;
20: }
```

SPACE FOR HINT

OUTPUT

```
C
is the
best
programming
language!!
```

The printf() Function

You can also display strings using the printf() library function. The printf() uses a format string and conversion specifiers to shape its output. To display a string, use the conversion specifier %s.

When printf() encounters a %s in its format string, the function matches the %s with the corresponding argument in its argument list. For a string, this argument must be a pointer to the string that you want displayed. The printf() function displays the string on-screen, stopping when it reaches the string's terminating null character. For example:

```
char *str = "A message to display";
printf("%s", str);
```

You can also display multiple strings and mix them with literal text and/or numeric variables:

```
char *bank = "First Federal";
char *name = "John Doe";
int balance = 1000;
printf("The balance at %s for %s is %d.", bank, name, balance);
```

The resulting output is

```
The balance at First Federal for John Doe is 1000.
```

SPACE FOR HINT

Reading Strings from the Keyboard

In addition to displaying strings, programs often need to accept inputted string data from the user via the keyboard. The C library has two functions that can be used for this purpose--gets() and scanf(). Before you can read in a string from the keyboard, however, you must have somewhere to put it.

Inputting Strings Using the gets() Function

The gets() function gets a string from the keyboard. When gets() is called, it reads all characters typed at the keyboard up to the first newline character (which you generate by pressing Enter). This function discards the newline, adds a null character, and gives the string to the calling program. The string is stored at the location indicated by a pointer to type char passed to gets(). A program that uses gets() must #include the file STDIO.H. Here is an example.

Using gets() to input string data from the keyboard.

```
1:  /* Demonstrates using the gets() library function. */
2:
3:  #include <stdio.h>
4:
5:  /* Allocate a character array to hold input. */
6:
7:  char input[81];
8:
9:  main()
10: {
11:     puts("Enter some text, then press Enter: ");
12:     gets(input);
13:     printf("You entered: %s\n", input);
14:
15:     return 0;
16: }
```

```
Enter some text, then press Enter:
This is a test
You entered: This is a test
```

Example : Using the gets() return value to test for the input of a blank line.

```
1:  /* Demonstrates using the gets() return value. */
2:
3:  #include <stdio.h>
4:
5:  /* Declare a character array to hold input, and a pointer. */
```

```
7:  char input[81], *ptr;
8:
9:  main()
10: {
11:     /* Display instructions. */
12:
13:     puts("Enter text a line at a time, then press Enter.");
14:     puts("Enter a blank line when done.");
15:
16:     /* Loop as long as input is not a blank line. */
17:
18:     while ( *(ptr = gets(input)) != NULL)
19:        printf("You entered %s\n", input);
20:
21:     puts("Thank you and good-bye\n");
22:
23:     return 0;
24: }
```

```
Enter text a line at a time, then press Enter.
Enter a blank line when done.
First string
You entered First string
Two
You entered Two
Bradley L. Jones
You entered Bradley L. Jones
Thank you and good-bye
```

The gets() Function

```
#include <stdio.h>
char *gets(char *str);
```

The gets() function gets a string, str, from the standard input device, usually the keyboard. The string consists of any characters entered until a newline character is read. At that point, a null is appended to the end of the string.

Then the gets() function returns a pointer to the string just read. If there is a problem getting the string, gets() returns null.

Example

```
/* gets() example */
#include <stdio.h>
char line[256];
void main()
{
  printf( "Enter a string:\n");
```

SPACE FOR HINT

SPACE FOR HINT

```
  gets( line );
  printf( "\nYou entered the following string:\n" );
  printf( "%s\n", line );
}
```

Inputting Strings Using the scanf() Function

Example : Inputting numeric and text data with scanf().

```
1:  /* Demonstrates using scanf() to input numeric and text data. */
2:
3:  #include <stdio.h>
4:
5:  char lname[81], fname[81];
6:  int count, id_num;
7:
8:  main()
9:  {
10:     /* Prompt the user. */
11:
12:     puts("Enter last name, first name, ID number separated");
13:     puts("by spaces, then press Enter.");
14:
15:     /* Input the three data items. */
16:
17:     count = scanf("%s%s%d", lname, fname, &id_num);
18:
19:     /* Display the data. */
20:
21:      printf("%d items entered: %s %s %d \n", count, fname, lname,
id_num);
22:
23:     return 0;
24: }
```

```
Enter last name, first name, ID number separated
by spaces, then press Enter.
Jones Bradley 12345
3 items entered: Bradley Jones 12345
```

C.Y.P

1. What is Array ?

2. Explain how to pass the array of functions.

3. Explain about Multidimensional array.

4. Write a program to demonstrate two-dimensional array

Table of Contents

SPACE FOR HINT

UNIT - 4

4. INTRODUCTION - STRUCTURE

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called ``records'' in some languages, notably Pascal.) Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities. A structure can contain any of C's data types, including arrays and other structures. The variables in a structure, unlike those in an array, can be of different variable types. Each variable within a structure is called a member of the structure.

One traditional example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc. Some of these in turn could be structures: a name has several components, as does an address and even a salary. Another example, more typical for C, comes from graphics: a point is a pair of coordinate, a rectangle is a pair of points, and so on.

5. Defining and Declaring Structure

If you're writing a graphics program, your code needs to deal with the coordinates of points on the screen. Screen coordinates are written as an x value, giving the horizontal position, and a y value, giving the vertical position. You can define a structure named coord that contains both the x and y values of a screen location as follows:

```
struct coord {
   int x;
   int y;
};
```

The struct keyword, which identifies the beginning of a structure definition, must be followed immediately by the structure name, or tag (which follows the same rules as other C variable names). Within the braces following the structure name is a list of the structure's member variables. You must give a variable type and name for each member.

The preceding statements define a structure type named coord that contains two integer variables, x and y. They do not, however, actually create any instances of the structure coord. In other words, they don't declare (set aside storage for) any structures. There are two ways to declare structures. One is to follow the structure definition with a list of one or more variable names, as is done here:

```
    int y;
} first, second;
```

These statements define the structure type coord and declare two structures, first and second, of type coord. first and second are each instances of type coord; first contains two integer members named x and y, and so does second.

This method of declaring structures combines the declaration with the definition. The second method is to declare structure variables at a different location in your source code from the definition. The following statements also declare two instances of type coord:

```
struct coord {
    int x;
    int y;
};
/* Additional code may go here */
struct coord first, second;
```

5.1 The struct Keyword – Processing a Structure

```
struct tag {
    structure_member(s);
    /* additional statements may go here */
} instance;
```

The struct keyword is used to declare structures. A structure is a collection of one or more variables (structure_members) that have been grouped under a single name for easy manipulation. The variables don't have to be of the same variable type, nor do they have to be simple variables. Structures also can hold arrays, pointers, and other structures.

The keyword struct identifies the beginning of a structure definition. It's followed by a tag that is the name given to the structure. Following the tag are the structure members, enclosed in braces. An instance, the actual declaration of a structure, can also be defined. If you define the structure without the instance, it's just a template that can be used later in a program to declare structures. Here is a template's format:

```
struct tag {
    structure_member(s);
    /* additional statements may go here */
};
```

To use the template, you use the following format:

```
struct tag instance;
```

SPACE FOR HINT

SPACE FOR HINT

To use this format, you must have previously declared a structure with the given tag.

Example 1

```
/* Declare a structure template called SSN */
struct SSN {
   int first_three;
   char dash1;
   int second_two;
   char dash2;
   int last_four;
}
/* Use the structure template */
struct SSN customer_ssn;
```

Example 2

```
/* Declare a structure and instance together */
struct date {
   char month[2];
   char day[2];
   char year[4];
} current_date;
```

Example 3

```
/* Declare and initialize a structure */
struct time {
   int hours;
   int minutes;
   int seconds;
} time_of_birth = { 8, 45, 0 };
```

5.2 Initializing Structure

Like other C variable types, structures can be initialized when they're declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values separated by commas and enclosed in braces. For example, look at the following statements:

```
1: struct sale {
2:    char customer[20];
3:    char item[20];
4:    float amount;
5: } mysale = { "Acme Industries",
6:              "Left-handed widget",
7:              1000.00
8:            };
```

SPACE FOR HINT

For a structure that contains structures as members, list the initialization values in order. They are placed in the structure members in the order in which the members are listed in the structure definition. Here's an example that expands on the previous one:

```
1: struct customer {
2:     char firm[20];
3:     char contact[25];
4: }
5:
6: struct sale {
7:    struct customer buyer;
8:    char item[20];
9:    float amount;
10: } mysale = { { "Acme Industries", "George Adams"},
11:              "Left-handed widget",
12:               1000.00
13:          };
```

You can also initialize arrays of structures. The initialization data that you supply is applied, in order, to the structures in the array. For example, to declare an array of structures of type sale and initialize the first two array elements (that is, the first two structures), you could write

```
1: struct customer {
2:     char firm[20];
3:     char contact[25];
4: };
5:
6: struct sale {
7:     struct customer buyer;
8:     char item[20];
9:     float amount;
10: };
11:
12:
13: struct sale y1990[100] = {
14:     { { "Acme Industries", "George Adams"},
15:        "Left-handed widget",
16:         1000.00
17:     }
18:     { { "Wilson & Co.", "Ed Wilson"},
19:        "Type 12 gizmo",
20:         290.00
21:     }
22: };
```

SPACE FOR HINT

5.3 Accessing Structure Members

Individual structure members can be used like other variables of the same type. Structure members are accessed using the structure member operator (.), also called the dot operator, between the structure name and the member name. Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, you could write

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, you could write

```
printf("%d,%d", second.x, second.y);
```

At this point, you might be wondering what the advantage is of using structures rather than individual variables. One major advantage is that you can copy information between structures of the same type with a simple equation statement. Continuing with the preceding example, the statement

```
first = second;
```

is equivalent to this statement:

```
first.x = second.x;
first.y = second.y;
```

When your program uses complex structures with many members, this notation can be a great time-saver. Other advantages of structures will become apparent as you learn some advanced techniques. In general, you'll find structures to be useful whenever information of different variable types needs to be treated as a group. For example, in a mailing list database, each entry could be a structure, and each piece of information (name, address, city, and so on) could be a structure member.

More-Complex Structures

Now that you have been introduced to simple structures, you can get to the more interesting and complex types of structures. These are structures that contain other structures as members and structures that contain arrays as members.

5.4 Structure that Contain Structure

SPACE FOR HINT

As mentioned earlier, a C structure can contain any of C's data types. For example, a structure can contain other structures. The previous example can be extended to illustrate this.

Assume that your graphics program needs to deal with rectangles. A rectangle can be defined by the coordinates of two diagonally opposite corners. You've already seen how to define a structure that can hold the two coordinates required for a single point. You need two such structures to define a rectangle. You can define a structure as follows (assuming, of course, that you have already defined the type coord structure):

```
struct rectangle {
   struct coord topleft;
   struct coord bottomrt;
};
```

This statement defines a structure of type rectangle that contains two structures of type coord. These two type coord structures are named topleft and bottomrt.

The preceding statement defines only the type rectangle structure. To declare a structure, you must then include a statement such as

```
struct rectangle mybox;
```

You could have combined the definition and declaration, as you did before for the type coord:

```
struct rectangle {
   struct coord topleft;
   struct coord bottomrt;
} mybox;
```

To access the actual data locations (the type int members), you must apply the member operator (.) twice. Thus, the expression

```
mybox.topleft.x
```

refers to the x member of the topleft member of the type rectangle structure named mybox. To define a rectangle with coordinates (0,10),(100,200), you would write

```
mybox.topleft.x = 0;
mybox.topleft.y = 10;
mybox.bottomrt.x = 100;
mybox.bottomrt.y = 200;
```

SPACE FOR HINT

Let's look at an example of using structures that contain other structures. The following program takes input from the user for the coordinates of a rectangle and then calculates and displays the rectangle's area. Note the program's assumptions, given in comments near the start of the program (lines 3 through 8).

Program : A demonstration of structures that contain other structures.

```
1:  /* Demonstrates structures that contain other structures. */
2:
3:  /* Receives input for corner coordinates of a rectangle and
4:     calculates the area. Assumes that the y coordinate of the
5:     upper-left corner is greater than the y coordinate of the
6:     lower-right corner, that the x coordinate of the lower-
7:     right corner is greater than the x coordinate of the upper-
8:     left corner, and that all coordinates are positive. */
9:
10: #include <stdio.h>
11:
12: int length, width;
13: long area;
14:
15: struct coord{
16:     int x;
17:     int y;
18: };
19:
20: struct rectangle{
21:     struct coord topleft;
22:     struct coord bottomrt;
23: } mybox;
24:
25: main()
26: {
27:     /* Input the coordinates */
28:
29:     printf("\nEnter the top left x coordinate: ");
30:     scanf("%d", &mybox.topleft.x);
31:
32:     printf("\nEnter the top left y coordinate: ");
33:     scanf("%d", &mybox.topleft.y);
34:
35:     printf("\nEnter the bottom right x coordinate: ");
36:     scanf("%d", &mybox.bottomrt.x);
37:
38:     printf("\nEnter the bottom right y coordinate: ");
39:     scanf("%d", &mybox.bottomrt.y);
40:
41:     /* Calculate the length and width */
```

```
42:
43:    width = mybox.bottomrt.x - mybox.topleft.x;
44:    length = mybox.bottomrt.y - mybox.topleft.y;
45:
46:    /* Calculate and display the area */
47:
48:    area = width * length;
49:    printf("\nThe area is %ld units.\n", area);
50:
51:    return 0;
52: }
```

```
Enter the top left x coordinate: 1
Enter the top left y coordinate: 1
Enter the bottom right x coordinate: 10
Enter the bottom right y coordinate: 10
The area is 81 units.
```

5. Structures That Contain Arrays

You can define a structure that contains one or more arrays as members. The array can be of any C data type (int, char, and so on). For example, the statements

```
struct data{
  int  x[4];
  char y[10];
};
```

define a structure of type data that contains a four-element integer array member named x and a 10-element character array member named y. You can then declare a structure named record of type data as follows:

```
struct data record;
```

You access individual elements of arrays that are structure members using a combination of the member operator and array subscripts:

```
record.x[2] = 100;
record.y[1] = 'x';
```

Character arrays are most frequently used to store strings. that the name of an array, without brackets, is a pointer to the array. Because this holds true for arrays that are structure members, the expression

```
record.y
```

is a pointer to the first element of array y[] in the structure record. Therefore, you could print the contents of y[] on-screen using the statement

SPACE FOR HINT

SPACE FOR HINT

puts(record.y);

The following program uses a structure that contains a type float variable and two type char arrays.

Program : A structure that contains array members.

```
1:  /* Demonstrates a structure that has array members. */
2:
3:  #include <stdio.h>
4:
5:  /* Define and declare a structure to hold the data. */
6:  /* It contains one float variable and two char arrays. */
7:
8:  struct data{
9:      float amount;
10:     char fname[30];
11:     char lname[30];
12: } rec;
13:
14: main()
15: {
16:     /* Input the data from the keyboard. */
17:
18:     printf("Enter the donor's first and last names,\n");
19:     printf("separated by a space: ");
20:     scanf("%s %s", rec.fname, rec.lname);
21:
22:     printf("\nEnter the donation amount: ");
23:     scanf("%f", &rec.amount);
24:
25:     /* Display the information. */
26:     /* Note: %.2f specifies a floating-point value */
27:     /* to be displayed with two digits to the right */
28:     /* of the decimal point. */
29:
30:     /* Display the data on the screen. */
31:
32:     printf("\nDonor %s %s gave $%.2f.\n", rec.fname, rec.lname,
33:             rec.amount);
34:
35:     return 0;
36: }
```

```
Enter the donor's first and last names,
separated by a space: Bradley Jones
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```

6.1. Arrays of Structures

SPACE FOR HINT

If you can have structures that contain arrays, can you also have arrays of structures? You bet you can! In fact, arrays of structures are very powerful programming tools. Here's how it's done.

You've seen how a structure definition can be tailored to fit the data your program needs to work with. Usually a program needs to work with more than one instance of the data. For example, in a program to maintain a list of phone numbers, you can define a structure to hold each person's name and number:

```
struct entry{
   char fname[10];
   char lname[12];
   char phone[8];
};
```

A phone list must hold many entries, however, so a single instance of the entry structure isn't of much use. What you need is an array of structures of type entry. After the structure has been defined, you can declare an array as follows:

```
struct entry list[1000];
```

This statement declares an array named list that contains 1,000 elements. Each element is a structure of type entry and is identified by subscript like other array element types. Each of these structures has three elements, each of which is an array of type char. This entire complex creation is diagrammed in Figure 11.3.

Figure 11.3. The organization of the array of structures defined in the text.

When you have declared the array of structures, you can manipulate the data in many ways. For example, to assign the data in one array element to another array element, you would write

```
list[1] = list[5];
```

This statement assigns to each member of the structure list[1] the values contained in the corresponding members of list[5]. You can also move data between individual structure members. The statement

```
strcpy(list[1].phone, list[5].phone);
```

copies the string in list[5].phone to list[1].phone. (The strcpy() library function copies one string to another string. If you want to, you can also move data between individual elements of the structure member arrays:

SPACE FOR HINT

```
list[5].phone[1] = list[2].phone[3];
```

This statement moves the second character of list[5]'s phone number to the fourth position in list[2]'s phone number. (Don't forget that subscripts start at offset 0.)

The following program demonstrates the use of arrays of structures. Moreover, it demonstrates arrays of structures that contain arrays as members.

Program : Arrays of structures.

```
1:  /* Demonstrates using arrays of structures. */
2:
3:  #include <stdio.h>
4:
5:  /* Define a structure to hold entries. */
6:
7:  struct entry {
8:      char fname[20];
9:      char lname[20];
10:     char phone[10];
11: };
12:
13: /* Declare an array of structures. */
14:
15: struct entry list[4];
16:
17: int i;
18:
19: main()
20: {
21:
22:     /* Loop to input data for four people. */
23:
24:     for (i = 0; i < 4; i++)
25:     {
26:         printf("\nEnter first name: ");
27:         scanf("%s", list[i].fname);
28:         printf("Enter last name: ");
29:         scanf("%s", list[i].lname);
30:         printf("Enter phone in 123-4567 format: ");
31:         scanf("%s", list[i].phone);
32:     }
33:
34:     /* Print two blank lines. */
35:
36:     printf("\n\n");
37:
38:     /* Loop to display data. */
```

```
39:
40:    for (i = 0; i < 4; i++)
41:    {
42:       printf("Name: %s %s", list[i].fname, list[i].lname);
43:       printf("\t\tPhone: %s\n", list[i].phone);
44:    }
45:
46:    return 0;
47: }
```

```
Enter first name: Bradley
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Peter
Enter last name: Aitken
Enter phone in 123-4567 format: 555-3434
Enter first name: Melissa
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Deanna
Enter last name: Townsend
Enter phone in 123-4567 format: 555-1234
Name: Bradley Jones          Phone: 555-1212
Name: Peter Aitken           Phone: 555-3434
Name: Melissa Jones          Phone: 555-1212
Name: Deanna Townsend        Phone: 555-1234
```

SPACE FOR HINT

7. Structure and Functions

The C language allows you to pass an entire structure to a function. In addition, a function can return a structure back to its caller.

To show you how to pass a structure to a function. Let us see the following example program.

Program : Passing a structure to a function.

```
1: /*  Passing a structure to a function */
2: #include <stdio.h>
3:
4: struct computer {
5:    float cost;
6:    int year;
7:    int cpu_speed;
8:    char cpu_type[16];
9: };
10: /* create synonym */
11: typedef struct computer SC;
12: /* function declaration */
```

SPACE FOR HINT

```
13: SC DataReceive(SC s);
14:
15: main(void)
16: {
17:    SC model;
18:
19:    model = DataReceive(model);
20:    printf("Here are what you entered:\n");
21:    printf("Year: %d\n", model.year);
22:    printf("Cost: $%6.2f\n", model.cost);
23:    printf("CPU type: %s\n", model.cpu_type);
24:    printf("CPU speed: %d MHz\n", model.cpu_speed);
25:
26:    return 0;
27: }
28: /* function definition */
29: SC DataReceive(SC s)
30: {
31:    printf("The type of the CPU inside your computer?\n");
32:       gets(s.cpu_type);
33:    printf("The speed(MHz) of the CPU?\n");
34:       scanf("%d", &s.cpu_speed);
35:    printf("The year your computer was made?\n");
36:       scanf("%d", &s.year);
37:    printf("How much you paid for the computer?\n");
38:       scanf("%f", &s.cost);
39:    return s;
40: }
```

OUTPUT

```
The type of the CPU inside your computer?
Pentium
The speed(MHz) of the CPU?
100
The year your computer was made?
1996
How much you paid for the computer?
1234.56
Here are what you entered:
Year: 1996
Cost: $1234.56
CPU type: Pentium
CPU speed: 100 MHz
C:\app>
```

Explanation

The purpose of the program is to show you how to pass a structure to a function. The structure with the tag name of computer, is declared in lines 4_9.

Note that in line 11 the typedef keyword is used to define a synonym, SC, for structure computer. Then SC is used in the sequential declarations.

The DataReceive() function is declared in line 13, with the structure of computer as its argument (that is, the synonym SC and the variable name s), so that a copy of the structure can be passed to the function.

In addition, the DataReceive() function returns the copy of the structure back to the caller after the content of the structure is updated. To do this, SC is prefixed to the function in line 13 to indicate the data type of the value returned by the function.

The statement in line 17 defines the structure model with SC. The DataReceive() function is passed with the name of the model structure in line 19, and then the value returned by the function is assigned back to model as well. Note that if the DataReceive() function return value is not assigned to model, the changes made to s in the function will not be evident in model.

The definition of the DataReceive() function is shown in lines 29_40, from which you can see that the new data values entered by the user are saved into the corresponding members of the structure that is passed to the function. At the end of the function, the copy of the updated structure is returned in line 39.

Then, back to the main() function of the program, lines 21_24 print out the updated contents held by the members of the structure.

SPACE FOR HINT

Example 1

Passing a structure as a function argument.

```
1: /* Demonstrates passing a structure to a function. */
2:
3: #include <stdio.h>
4:
5: /* Declare and define a structure to hold the data. */
6:
7: struct data{
8:     float amount;
9:     char fname[30];
10:    char lname[30];
11: } rec;
12:
13: /* The function prototype. The function has no return value, */
```

SPACE FOR HINT

```
14: /* and it takes a structure of type data as its one argument. */
15:
16: void print_rec(struct data x);
17:
18: main()
19: {
20:     /* Input the data from the keyboard. */
21:
22:     printf("Enter the donor's first and last names,\n");
23:     printf("separated by a space: ");
24:     scanf("%s %s", rec.fname, rec.lname);
25:
26:     printf("\nEnter the donation amount: ");
27:     scanf("%f", &rec.amount);
28:
29:     /* Call the display function. */
30:     print_rec( rec );
31:
32:     return 0;
33: }
34: void print_rec(struct data x)
35: {
36:     printf("\nDonor %s %s gave $%.2f.\n", x.fname, x.lname,
37:             x.amount);
38: }
```

```
Enter the donor's first and last names,
separated by a space: Bradley Jones
Enter the donation amount: 1000.00
Donor Bradley Jones gave $1000.00.
```

7.1 Functions with structures

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with &, and accessing its members. Copy and assignment include passing arguments to functions and returning values from functions as well. Structures may not be compared. A structure may be initialized by a list of constant member values; an automatic structure may also be initialized by an assignment.

Let us investigate structures by writing some functions to manipulate points and rectangles. There are at least three possible approaches: pass components separately, pass an entire structure, or pass a pointer to it. Each has its good points and bad points.

The first function, makepoint, will take two integers and return a point structure:

```
/* ma[illegible] from [illegible]nents */
```

```
{
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

SPACE FOR HINT

Notice that there is no conflict between the argument name and the member with the same name; indeed the re-use of the names stresses the relationship. makepoint can now be used to initialize any structure dynamically, or to provide structure arguments to a function:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);
screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,(screen.pt1.y +
screen.pt2.y)/2);
```

The next step is a set of functions to do arithmetic on points. For instance,

```
/* addpoints:  add two points */
struct addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Here both the arguments and the return value are structures. We incremented the components in p1 rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others.

As another example, the function ptinrect tests whether a point is inside a rectangle, where we have adopted the convention that a rectangle includes its left and bottom sides but not its top and right sides:

```
/* ptinrect:  return 1 if p in r, 0 if not */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
        && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

This assumes that the rectangle is presented in a standard form where the pt1 coordinates are less than the pt2 coordinates. The following function returns a rectangle guaranteed to be in canonical form:

SPACE FOR HINT

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
/* canonrect: canonicalize coordinates of rectangle */
struct rect canonrect(struct rect r)
{
   struct rect temp;
   temp.pt1.x = min(r.pt1.x, r.pt2.x);
   temp.pt1.y = min(r.pt1.y, r.pt2.y);
   temp.pt2.x = max(r.pt1.x, r.pt2.x);
   temp.pt2.y = max(r.pt1.y, r.pt2.y);
   return temp;
}
```

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. Structure pointers are just like pointers to ordinary variables. The declaration

```
struct point *pp;
```

says that pp is a pointer to a structure of type struct point. If pp points to a point structure, *pp is the structure, and (*pp).x and (*pp).y are the members. To use pp, we might write, for example,

```
struct point origin, *pp;
pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

The parentheses are necessary in (*pp).x because the precedence of the structure member operator . is higher then *. The expression *pp.x means *(pp.x), which is illegal here because x is not a pointer.

Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then p->member-of-structure refers to the particular member. So we could write instead

```
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

Both . and -> associate from left to right, so if we have

```
struct rect r, *rp = &r;
```

then these four expressions are equivalent:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

The structure operators . and -> together with () for function calls and [] for subscripts are at the top of the precedence hierarchy and thus bind

SPACE FOR HINT

```
  struct {
     int len;
     char *str;
  } *p;
then
  ++p->len
```

increments len, not p, because the implied parenthesization is ++(p->len). Parentheses can be used to alter binding: (++p)->len increments p before accessing len, and (p++)->len increments p afterward. (This last set of parentheses is unnecessary.)

In the same way, *p->str fetches whatever str points to; *p->str++ increments str after accessing whatever it points to (just like *s++); (*p->str)++ increments whatever str points to; and *p++->str increments p after accessing whatever str points to.

8. Self-referential Structures

Suppose we want to handle the more general problem of counting the occurrences of all the words in some input. Since the list of words isn't known in advance, we can't conveniently sort it and use a binary search. Yet we can't do a linear search for each word as it arrives, to see if it's already been seen; the program would take too long. (More precisely, its running time is likely to grow quadratically with the number of input words.) How can we organize the data to copy efficiently with a list or arbitrary words?

One solution is to keep the set of words seen so far sorted at all times, by placing each word into its proper position in the order as it arrives. This shouldn't be done by shifting words in a linear array, though - that also takes too long. Instead we will use a data structure called a binary tree.

The tree contains one ``node'' per distinct word; each node contains

A pointer to the text of the word,
A count of the number of occurrences,
A pointer to the left child node,
A pointer to the right child node.

No node may have more than two children; it might have only zero or one.

The nodes are maintained so that at any node the left subtree contains only words that are lexicographically less than the word at the node, and the right subtree contains only words that are greater. This is the tree for the sentence ``now is the time for all good men to come to the aid of their party'', as built by inserting each word as it is encountered:

SPACE FOR HINT

To find out whether a new word is already in the tree, start at the root and compare the new word to the word stored at that node. If they match, the question is answered affirmatively. If the new record is less than the tree word, continue searching at the left child, otherwise at the right child. If there is no child in the required direction, the new word is not in the tree, and in fact the empty slot is the proper place to add the new word. This process is recursive, since the search from any node uses a search from one of its children. Accordingly, recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is most conveniently represented as a structure with four components:

```
struct tnode {    /* the tree node: */
   char *word;        /* points to the text */
   int count;         /* number of occurrences */
   struct tnode *left;  /* left child */
   struct tnode *right;  /* right child */
};
```

This recursive declaration of a node might look chancy, but it's correct. It is illegal for a structure to contain an instance of itself, but

```
struct tnode *left;
```

declares left to be a pointer to a tnode, not a tnode itself.

Occasionally, one needs a variation of self-referential structures: two structures that refer to each other. The way to handle this is:

```
struct t {
   ...
   struct s *p;   /* p points to an s */
};
struct s {
   ...
   struct t *q;   /* q points to a t */
};
```

The code for the whole program is surprisingly small, given a handful of supporting routines like getword that we have already written. The main routine reads words with getword and installs them in the tree with addtree.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
```

```
int getword(char *, int);
/* word frequency count */
main()
{
   struct tnode *root;
   char word[MAXWORD];
   root = NULL;
   while (getword(word, MAXWORD) != EOF)
      if (isalpha(word[0]))
         root = addtree(root, word);
   treeprint(root);
   return 0;
}
```

SPACE FOR HINT

The function addtree is recursive. A word is presented by main to the top level (the root) of the tree. At each stage, that word is compared to the word already stored at the node, and is percolated down to either the left or right subtree by a recursive call to adtree. Eventually, the word either matches something already in the tree (in which case the count is incremented), or a null pointer is encountered, indicating that a node must be created and added to the tree. If a new node is created, addtree returns a pointer to it, which is installed in the parent node.

```
struct tnode *talloc(void);
char *strdup(char *);
/* addtree:  add a node with w, at or below p */
struct treenode *addtree(struct tnode *p, char *w)
{
   int cond;
   if (p == NULL) {     /* a new word has arrived */
      p = talloc();    /* make a new node */
      p->word = strdup(w);
      p->count = 1;
      p->left = p->right = NULL;
   } else if ((cond = strcmp(w, p->word)) == 0)
      p->count++;      /* repeated word */
   else if (cond < 0)   /* less than into left subtree */
      p->left = addtree(p->left, w);
   else            /* greater than into right subtree */
      p->right = addtree(p->right, w);
   return p;
}
```

Storage for the new node is fetched by a routine talloc, which returns a pointer to a free space suitable for holding a tree node, and the new word is copied nto a hidden space by strdup. (We will discuss these routines in a moment.) The count is initialized, and the two children are made null. This part of the code is executed only at the leaves of the tree, when a new node is being added. We have (unwisely) omitted error checking on the values returned by strdup and talloc.

SPACE FOR HINT

treeprint prints the tree in sorted order; at each node, it prints the left subtree (all the words less than this word), then the word itself, then the right subtree (all the words greater). If you feel shaky about how recursion works, simulate treeprint as it operates on the tree shown above.

```
/* treeprint:  in-order print of tree p */
void treeprint(struct tnode *p)
{
   if (p != NULL) {
      treeprint(p->left);
      printf("%4d %s\n", p->count, p->word);
      treeprint(p->right);
   }
}
```

A practical note: if the tree becomes ``unbalanced" because the words don't arrive in random order, the running time of the program can grow too much. As a worst case, if the words are already in order, this program does an expensive simulation of linear search. There are generalizations of the binary tree that do not suffer from this worst-case behavior, but we will not describe them here.

9. Bit-fields

When storage space is at a premium, it may be necessary to pack several objects into a single machine word; one common use is a set of single-bit flags in applications like compiler symbol tables. Externally-imposed data formats, such as interfaces to hardware devices, also often require the ability to get at pieces of a word.

Imagine a fragment of a compiler that manipulates a symbol table. Each identifier in a program has certain information associated with it, for example, whether or not it is a keyword, whether or not it is external and/or static, and so on. The most compact way to encode such information is a set of one-bit flags in a single char or int.

The usual way this is done is to define a set of ``masks" corresponding to the relevant bit positions, as in

```
  #define KEYWORD  01
  #define EXTRENAL 02
  #define STATIC   04
```

or

```
  enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

The numbers must be powers of two. Then accessing the bits becomes a matter of ``bit-fiddling" with the shifting, masking, and complementing operators.

SPACE FOR HINT

Certain idioms appear frequently:

```
flags |= EXTERNAL | STATIC;
```

turns on the EXTERNAL and STATIC bits in flags, while

```
flags &= ~(EXTERNAL | STATIC);
```

turns them off, and

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

is true if both bits are off.

Although these idioms are readily mastered, as an alternative C offers the capability of defining and accessing fields within a word directly rather than by bitwise logical operators. A bit-field, or field for short, is a set of adjacent bits within a single implementation-defined storage unit that we will call a ``word.'' For example, the symbol table #defines above could be replaced by the definition of three fields:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

This defines a variable table called flags that contains three 1-bit fields. The number following the colon represents the field width in bits. The fields are declared unsigned int to ensure that they are unsigned quantities.

Individual fields are referenced in the same way as other structure members: flags.is_keyword, flags.is_extern, etc. Fields behave like small integers, and may participate in arithmetic expressions just like other integers. Thus the previous examples may be written more naturally as

```
flags.is_extern = flags.is_static = 1; to turn the bits on;

flags.is_extern = flags.is_static = 0; to turn them off; and

if (flags.is_extern == 0 && flags.is_static == 0)
...
```

to test them.

Almost everything about fields is implementation-dependent. Whether a field may overlap a word boundary is implementation-defined. Fields need not be names; unnamed fields (a colon and width only) are used for padding. The special width 0 may be used to force alignment at the next word boundary.

Fields are assigned left to right on some machines and right to left on others. This means that although fields are useful for maintaining internally-defined data structures, the question of which end comes first has to be carefully considered when picking apart externally-defined

SPACE FOR HINT

data; programs that depend on such things are not portable. Fields may be declared only as ints; for portability, specify signed or unsigned explicitly. They are not arrays and they do not have addresses, so the & operator cannot be applied on them.

10. Unions

Unions are derived data types, the way structures are. But unions have the same relationship to structures that you might have with a distant cousin who resembled you but turned out to be smuggling contraband in mexico. That is unions and structures look alike, but are engaged in totally different enterprises.

Both structures and unions are used to group a number of different variables together. But while a structure enables us to test a number of different variables stored at different places in memory as a number of union enables us to tartest the same space in memory as a number of different variables, that is, a union offers a way for a section of memory to be treated as a variable of one type on one occasion, and as a different variable of a different type of another occasion.

10.1 The union Keyword

```
union tag {
   union_member(s);
   /* additional statements may go here */
}instance;
```

The union keyword is used for declaring unions. A union is a collection of one or more variables (union_members) that have been grouped under a single name. In addition, each of these union members occupies the same area of memory.

The keyword union identifies the beginning of a union definition. It's followed by a tag that is the name given to the union. Following the tag are the union members enclosed in braces. An instance, the actual declaration of a union, also can be defined. If you define the structure without the instance, it's just a template that can be used later in a program to declare structures. The following is a template's format:

```
union tag {
   union_member(s);
   /* additional statements may go here */
};
```

To use the template, you would use the following format:

```
union tag instance;
```

To use this format, you must have previously declared a union with the given tag.

SPACE FOR HINT

Example 1

```
/* Declare a union template called tag */
union tag {
   int nbr;
   char character;
}
/* Use the union template */
union tag mixed_variable;
```

Example 2

```
/* Declare a union and instance together */
union generic_type_tag {
   char c;
   int i;
   float f;
   double d;
} generic;
```

Example 3

```
/* Initialize a union. */
union date_tag {
   char full_date[9];
   struct part_date_tag {
      char month[2];
      char break_value1;
      char day[2];
      char break_value2;
      char year[2];
   } part_date;
}date = {"01/01/97"};
```

The following program shows a more practical use of a union. Although this use is simplistic, it's one of the more common uses of a union.

Program : A practical use of a union.

```
1:  /* Example of a typical use of a union */
2:
3:  #include <stdio.h>
4:
5:  #define CHARACTER   'C'
6:  #define INTEGER     'I'
7:  #define FLOAT       'F'
8:
9:  struct generic_tag{
10:     char type;
```

SPACE FOR HINT

```
11:     union shared_tag {
12:        char   c;
13:        int    i;
14:        float  f;
15:     } shared;
16: };
17:
18: void print_function( struct generic_tag generic );
19:
20: main()
21: {
22:     struct generic_tag var;
23:
24:     var.type = CHARACTER;
25:     var.shared.c = `$';
26:     print_function( var );
27:
28:     var.type = FLOAT;
29:     var.shared.f = (float) 12345.67890;
30:     print_function( var );
31:
32:     var.type = `x';
33:     var.shared.i = 111;
34:     print_function( var );
35:     return 0;
36: }
37: void print_function( struct generic_tag generic )
38: {
39:     printf("\n\nThe generic value is...");
40:     switch( generic.type )
41:     {
42:        case CHARACTER: printf("%c",  generic.shared.c);
43:                        break;
44:        case INTEGER:   printf("%d",  generic.shared.i);
45:                        break;
46:        case FLOAT:     printf("%f",  generic.shared.f);
47:                        break;
48:        default:        printf("an unknown type: %c\n",
49:                           generic.type);
50:                        break;
51:     }
52: }
```

```
The generic value is...$
The generic value is...12345.678711
The generic value is...an unknown type: x
```

You might wonder why it would be necessary to do such a thing, but we will be seeing several very practical applications of unions soon. First let us take a look at simple example:

SPACE FOR HINT

```
\* Demo of union at work *\
main ()
{
        union a
        {
                int ;
                char ch [2];
        };
        union a key ;

        key. i =512;
        printf ("\nkey.i = %d", key.i );
        printf ("\nkey.ch[0] = %d", key.ch [0]);
        printf ("\nney.ch[1] = %d", key.ch [1]);
}
```

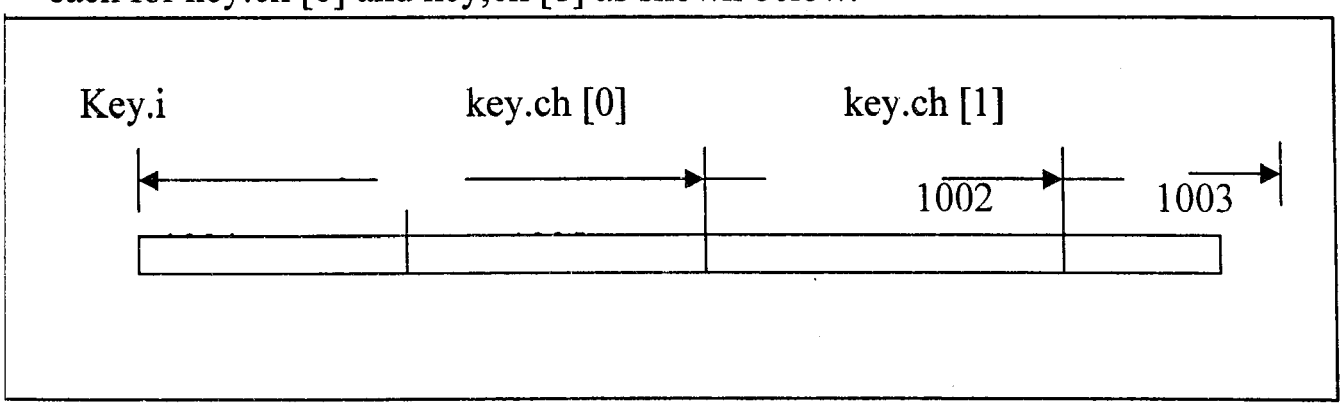
and here is the output.....

```
Key.i = 512
Key.ch[0] = 0
Key.ch[1] = 2
```

As you can see, first we declared a data type of the type union a and then a variable key to be of the type union. This is similar to the way we declare first the structure type and then the structure variables . also, the union the elements of the are accessed exactly the same way in which the structure elements are accessed, using a '.' Operation the following data types:

```
Struct a
{
        int I;
        char. Ch[2];
};
struct a key;
```

this data type would occupy 4 bytes in memory, 2 for key .i and one each for key.ch [0] and key,ch [1] as shown below.

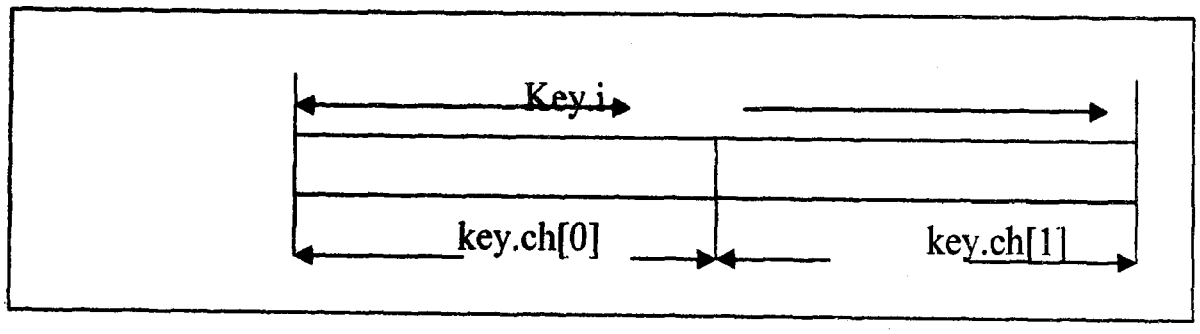


SPACE FOR HINT

Now we declare a similar data types, but instead of using a structure we use a union...

```
union a
{
        int i;
        char.ch [2];
}
union a key;
```

representation of this data type of in memory is shown below:



As shown in figure the union occupies only 2 bytes in memory. Note that the same memory locations which are used for key.i are also being by key.ch [0] and key.ch [1] means that the memo locations used by key.i cal also be accessed using key.ch[0] and key.ch [I]. What purpose does this serve? Well, know we can access the two bytes simultaneously (by using key.i) or that same two bytes individually using key .ch [0] and key.ch [1].

This is a frequent requirement while interacting with the hardware i.e. sometimes we are required to access two bytes simultaneously and sometimes each byte individually faced with a such a situation, usir union is the answer, usually.

One last thing. We cant assign different values to the different union elements at the same time. I.e if we assign a value to key.i it gets automatically assigned to key.ch [0] and key.ch[1] vice versa if assign a value of key.ch [0] or key .ch[1] it is bound to get assigned to key.i. here is a program which illustrates this fact.

```
main()
{
        union a
        {
                int i;
                char ch[2];
        };
        union a key;
        key.i =512;
        printf ("\nkey.i = %d', key .i );
```

SPACE FOR HINT

```
            printf ("\nkey.ch [0] = %d', key .ch[0] );
            printf ("\nkey.ch [1] = %d', key .ch[1]);

            key.ch [0] = 50 ; /* assign a new bvalue to key .ch [0] */
            printf ("\nkey.i = %d', key .i );
            printf ("\nkey.ch [0] = %d', key .ch[0] );
            printf ("\nkey.ch [1] = %d', key .ch[1]);
    }
```

and here is the output.....

```
key .i = 512
key .ch[0] =0
key .ch[1] = 2
key .i = 562
key .ch [0] = 50
key .ch [1] = 2
```

Let us reiterate that a union provides a way to look at the same data in several different ways. For example, there can exist a union as shown below.

```
union b
{
        double d;
        float f [2];
        int I [4];
        char ch [8];
};
union b data;
```

in what different ways can the data be accessed from it? Sometimes as a complete set of eight bytes (b.d) sometimes as two sets of 4 bytes eachg (b.f [0] and b.f [1]) sometimes as four sets of 2 bytes each (b.i[0], b.i [1], b.i [2] and b.i [3]) and sometimes as eight sets of 1 byte each (b.ch [0], b.ch[1].....b.ch [7]).

10.2 Accessing Union Members

Individual union members can be used in the same way that structure members can be used--by using the member operator (.). However, there is an important difference in accessing union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it's important to access only one member at a time. Here we presents an example.

Program : An example of the wrong use of unions.

```
1:  /* Example of using more than one union member at a time */
2:  #include <stdio.h>
3:
```

SPACE FOR HINT

```
4:  main()
5:  {
6:      union shared_tag {
7:          char   c;
8:          int    i;
9:          long   l;
10:         float  f;
11:         double d;
12:     } shared;
13:
14:     shared.c = `$';
15:
16:     printf("\nchar c   = %c",  shared.c);
17:     printf("\nint i    = %d",  shared.i);
18:     printf("\nlong l   = %ld", shared.l);
19:     printf("\nfloat f  = %f",  shared.f);
20:     printf("\ndouble d = %f",  shared.d);
21:
22:     shared.d = 123456789.8765;
23:
24:     printf("\n\nchar c   = %c",  shared.c);
25:     printf("\nint i    = %d",  shared.i);
26:     printf("\nlong l   = %ld", shared.l);
27:     printf("\nfloat f  = %f",  shared.f);
28:     printf("\ndouble d = %f\n",  shared.d);
29:
30:     return 0;
31: }
```

```
char c   = $
int i    = 4900
long l   = 437785380
float f  = 0.000000
double d = 0.000000

char c   = 7
int i    = -30409
long l   = 1468107063
float f  = 284852666499072.000000
double d = 123456789.876500
```

11. Enumerations

An *enumeration* is a set of named integer constants that specify all the legal values a variable of that type may have. Enumerations are common in everyday life. For example, an enumeration of the coins used in the United States is

penny, nickel, dime, quarter, half-dollar, dollar

SPACE FOR HINT

Enumerations are defined much like structures; the keyword **enum** signals the start of an enumeration type.

11.1. Defining Enumeration

The general form for enumerations is

enum *enum-type-name* { *enumeration list* } *variable_list*;

Here, both the type name and the variable list are optional. (But at least one must be present.) The following code fragment defines an enumeration called **coin**:

```
enum coin { penny, nickel, dime, quarter,half_dollar, dollar};
```

The enumeration type name can be used to declare variables of its type. In C, the following declares **money** to be a variable of type **coin**.

```
enum coin money;
```

In C++, the variable **money** may be declared using this shorter form:

```
coin money;
```

Note

In C++, an enumeration name specifies a complete type. In C, an enumeration name is
its tag and it requires the keyword **enum** to complete it. (This is similar to the situation as it applies to structures and unions)

Given these declarations, the following types of statements are perfectly valid:

```
money = dime;

if(money==quarter) printf("Money is a quarter.\n");
```

The key point to understand about an enumeration is that each of the symbols stands for an integer value. As such, they may be used anywhere that an integer may be used. Each symbol is given a value one greater than the symbol that precedes it.

The value of the first enumeration symbol is 0. Therefore,

```
printf("%d %d", penny, dime);
```

displays **0 2** on the screen.

You can specify the value of one or more of the symbols by using an
[illegible] Do this by following the [illegible] with an equal sign and an

SPACE FOR HINT

integer value. Symbols that appear after initializers are assigned values greater than the previous initialization value.

For example, the following code assigns the value of 100 to **quarter**:

```
enum coin { penny, nickel, dime, quarter=100, half_dollar, dollar};
```

Now, the values of these symbols are
penny 0
nickel 1
dime 2
quarter 100
half_dollar 101
dollar 102

One common but erroneous assumption about enumerations is that the symbols can be input and output directly. This is not the case. For example, the following code fragment will not perform as desired:

```
/* this will not work */
money = dollar;
printf("%s", money);
```

Remember, **dollar** is simply a name for an integer; it is not a string. For the same reason, you cannot use this code to achieve the desired results:

```
/* this code is wrong */
strcpy(money, "dime");
```

That is, a string that contains the name of a symbol is not automatically converted to that symbol. Actually, creating code to input and output enumeration symbols is quite tedious (unless you are willing to settle for their integer values).

For example, you need the following code to display, in words, the kind of coins that **money** contains:

```
switch(money)
{
case penny: printf("penny");
break;
case nickel: printf("nickel");
break;
case dime: printf("dime");
break;
case quarter: printf("quarter");
break;
```

SPACE FOR HINT

```
}
```

Sometimes you can declare an array of strings and use the enumeration value as an index to translate that value into its corresponding string. For example, this code also outputs the proper string:

```
char name[][12]=
{
"penny",
"nickel",
"dime",
"quarter",
"half_dollar",
"dollar"
};
printf("%s", name[money]);
```

Of course, this only works if no symbol is initialized, because the string array must be indexed starting at 0. Since enumeration values must be converted manually to their human-readable string values for I/O operations, they are most useful in routines that do not make such conversions.

An enumeration is often used to define a compiler's symbol table, for example. Enumerations are also used to help prove the validity of a program by providing a compile-time redundancy check confirming that a variable is assigned only valid values.

Example 1

```
1:  /* Defining enum data types */
2:  #include <stdio.h>
3:  /* main() function */
4:  main()
5:  {
6:    enum language {human=100,
7:                   animal=50,
8:                   computer};
9:    enum days{SUN,
10:             MON,
11:             TUE,
12:             WED,
13:             THU,
14:             FRI,
15:             SAT};
16:
17:   printf("human: %d, animal: %d, computer: %d\n",
18:     human, animal, computer);
19:   printf("SUN: %d\n", SUN);
20:   printf("MON: %d\n", MON);
```

SPACE FOR HINT

```
21:   printf("TUE: %d\n", TUE);
22:   printf("WED: %d\n", WED);
23:   printf("THU: %d\n", THU);
24:   printf("FRI: %d\n", FRI);
25:   printf("SAT: %d\n", SAT);
26:
27:   return 0;
28: }
```

OUTPUT

```
human: 100,  animal: 50,  computer: 51
SUN: 0
MON: 1
TUE: 2
WED: 3
THU: 4
FRI: 5
SAT: 6
```

C.Y.P

1. What is Structure ?
2. Explain the steps to creating a Structure.
3. What is the difference between Structure and Function ?
4. What is Unions
5. What is Enumerations

TABLE OF CONTENTS

SPACE FOR HINT

UNIT - 5

5. INTRODUCTION - FILE

Most of the programs you write will use disk files for one purpose or another: data storage, configuration information, and so on. C performs all input and output, including disk files, by means of streams. You saw how to use C's predefined streams that are connected to specific devices such as the keyboard, screen, and (on DOS systems) the printer. Disk file streams work essentially the same way. This is one of the advantages of stream input/output--techniques for using one stream can be used with little or no change for other streams.

The major difference with disk file streams is that your program must explicitly create a stream associated with a specific disk file.

Types of Disk Files

Text streams are associated with text-mode files. Text-mode files consist of a sequence of lines. Each line contains zero or more characters and ends with one or more characters that signal end-of-line. The maximum line length is 255 characters.

Binary streams are associated with binary-mode files. Any and all data is written and read unchanged, with no separation into lines and no use of end-of-line characters. The NULL and end-of-line characters have no special significance and are treated like any other byte of data.

Some file input/output functions are restricted to one file mode, whereas other functions can use either mode. This chapter teaches you which mode to use with which functions.

6. File Operations

Every disk file has a name, and you must use filenames when dealing with disk files. Filenames are stored as strings, just like other text data. The rules as to what is acceptable for filenames and what is not differ from one operating system to another.

In DOS and Windows 3.x, a complete filename consists of a name that has from one to eight characters, optionally followed by a period and an extension that has from one to three characters. In contrast, the Windows 95 and Windows NT operating systems, as well as most UNIX systems, permit filenames up to 256 characters long.

Operating systems also differ in the characters that are permitted in filenames. In Windows 95, for example, the following characters are not permitted:

/ \ : * ? " < > |

SPACE FOR HINT

You must be aware of the filename rules of whichever operating system you're writing for.

6.1 Opening a File

The process of creating a stream linked to a disk file is called opening the file. When you open a file, it becomes available for reading (meaning that data is input from the file to the program), writing (meaning that data from the program is saved in the file), or both. When you're done using the file, you must close it. Closing a file.

To open a file, you use the fopen() library function. The prototype of fopen() is located in STDIO.H and reads as follows:

FILE *fopen(const char *filename, const char *mode);

This prototype tells you that fopen() returns a pointer to type FILE, which is a structure declared in STDIO.H. The members of the FILE structure are used by the program in the various file access operations, but you don't need to be concerned about them.

However, for each file that you want to open, you must declare a pointer to type FILE. When you call fopen(), that function creates an instance of the FILE structure and returns a pointer to that structure. You use this pointer in all subsequent operations on the file. If fopen() fails, it returns NULL. Such a failure could be caused, for example, by a hardware error or by trying to open a file on a diskette that hasn't been formatted.

Table :Values of mode for the fopen() function.

Mode	Meaning
r	Opens the file for reading. If the file doesn't exist, fopen() returns NULL.
w	Opens the file for writing. If a file of the specified name doesn't exist, it is created. If a file of the specified name does exist, it is deleted without warning, and a new, empty file is created.
a	Opens the file for appending. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is appended to the end of the file.
r+	Opens the file for reading and writing. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is added to the beginning of the file, overwriting existing data.

SPACE FOR HINT

w+ Opens the file for reading and writing. If a file of the specified name doesn't exist, it is created. If the file does exist, it is overwritten.

a+ Opens a file for reading and appending. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is appended to the end of the file.

The default file mode is text. To open a file in binary mode, you append a b to the mode argument. Thus, a mode argument of a would open a text-mode file for appending, whereas ab would open a binary-mode file for appending.

Remember that fopen() returns NULL if an error occurs. Error conditions that can cause a return value of NULL include the following:

1. Using an invalid filename.

2. Trying to open a file on a disk that isn't ready (the drive door isn't closed or the disk isn't formatted, for example).

3. Trying to open a file in a nonexistent directory or on a nonexistent disk drive.

4. Trying to open a nonexistent file in mode r.

Example program : using fopen() to open disk files in various modes.

```
1: /* Demonstrates the fopen() function. */
2: #include <stdlib.h>
3: #include <stdio.h>
4:
5: main()
6: {
7:    FILE *fp;
8:    char filename[40], mode[4];
9:
10:   while (1)
11:   {
12:
13:      /* Input filename and mode. */
14:
15:      printf("\nEnter a filename: ");
16:      gets(filename);
17:      printf("\nEnter a mode (max 3 characters): ");
18:      gets(mode);
19:
20:      /* Try to open the file. */
21:
```

```
22:       if ( (fp = fopen( filename, mode )) != NULL )
23:       {
24:          printf("\nSuccessful opening %s in mode %s.\n",
25:                filename, mode);
26:          fclose(fp);
27:          puts("Enter x to exit, any other to continue.");
28:          if ( (getc(stdin)) == `x')
29:             break;
30:          else
31:             continue;
32:       }
33:       else
34:       {
35:          fprintf(stderr, "\nError opening file %s in mode %s.\n",
36:                filename, mode);
37:          puts("Enter x to exit, any other to try again.");
38:          if ( (getc(stdin)) == `x')
39:             break;
40:          else
41:             continue;
42:       }
43:    }
44: }
```

```
Enter a filename: junk.txt
Enter a mode (max 3 characters): w
Successful opening junk.txt in mode w.
Enter x to exit, any other to continue.
j

Enter a filename: morejunk.txt
Enter a mode (max 3 characters): r
Error opening morejunk.txt in mode r.
Enter x to exit, any other to try again.
x
```

6.2 Writing and Reading File Data

A program that uses a disk file can write data to a file, read data from a file, or a combination of the two. You can write data to a disk file in three ways:

You can use formatted output to save formatted data to a file. You should use formatted output only with text-mode files. The primary use of formatted output is to create files containing text and numeric data to be read by other programs such as spreadsheets or databases. You rarely, if ever, use formatted output to create a file to be read again by a C program.

SPACE FOR HINT

SPACE FOR HINT

You can use character output to save single characters or lines of characters to a file. Although technically it's possible to use character output with binary-mode files, it can be tricky. You should restrict character-mode output to text files. The main use of character output is to save text (but not numeric) data in a form that can be read by C, as well as other programs such as word processors.

You can use direct output to save the contents of a section of memory directly to a disk file. This method is for binary files only. Direct output is the best way to save data for later use by a C program.

When you want to read data from a file, you have the same three options: formatted input, character input, or direct input. The type of input you use in a particular case depends almost entirely on the nature of the file being read. Generally, you will read data in the same mode that it was saved in, but this is not a requirement. However, reading a file in a mode different from the one it was written in requires a thorough knowledge of C and file formats.

The previous descriptions of the three types of file input and output suggest tasks best suited for each type of output. This is by no means a set of strict rules. The C language is very flexible (this is one of its advantages!), so a clever programmer can make any type of file output suit almost any need. As a beginning programmer, it might make things easier if you follow these guidelines, at least initially.

7. Formatted File Input and Output

Formatted file input/output deals with text and numeric data that is formatted in a specific way. It is directly analogous to formatted keyboard input and screen output done with the printf() and scanf() functions

7.1 Formatted File Output

Formatted file output is done with the library function fprintf(). The prototype of fprintf() is in the header file STDIO.H, and it reads as follows:

```
int fprintf(FILE *fp, char *fmt, ...);
```

The first argument is a pointer to type FILE. To write data to a particular disk file, you pass the pointer that was returned when you opened the file with fopen().

The second argument is the format string. The format string used by fprintf() follows exactly the same rules as printf().

The final argument is .... What does that mean? In a function prototype, ellipses represent a variable number of arguments. In other words, in

addition to the file pointer and the format string arguments, fprintf() takes zero, one, or more additional arguments. This is just like printf(). These arguments are the names of the variables to be output to the specified stream.

SPACE FOR HINT

Remember, fprintf() works just like printf(), except that it sends its output to the stream specified in the argument list. In fact, if you specify a stream argument of stdout, fprintf() is identical to printf().

Listing : The equivalence of fprintf() formatted output to both a file and to stdout.

```
1:  /* Demonstrates the fprintf() function. */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  void clear_kb(void);
6:
7:  main()
8:  {
9:     FILE *fp;
10:    float data[5];
11:    int count;
12:    char filename[20];
13:
14:    puts("Enter 5 floating-point numerical values.");
15:
16:    for (count = 0; count < 5; count++)
17:       scanf("%f", &data[count]);
18:
19:    /* Get the filename and open the file. First clear stdin */
20:    /* of any extra characters. */
21:
22:    clear_kb();
23:
24:    puts("Enter a name for the file.");
25:    gets(filename);
26:
27:    if ( (fp = fopen(filename, "w")) == NULL)
28:    {
29:       fprintf(stderr, "Error opening file %s.", filename);
30:       exit(1);
31:    }
32:
33:    /* Write the numerical data to the file and to stdout. */
34:
35:    for (count = 0; count < 5; count++)
36:    {
37:       fprintf(fp, "\ndata[%d] = %f", count, data[count]);
38:       fprintf(stdout, "\ndata[%d] = %f", count, data[count]);
```

SPACE FOR HINT

```
39:    }
40:    fclose(fp);
41:    printf("\n");
42:    return(0);
43: }
44:
45: void clear_kb(void)
46: /* Clears stdin of any waiting characters. */
47: {
48:    char junk[80];
49:    gets(junk);
50: }
```

```
Enter 5 floating-point numerical values.
3.14159
9.99
1.50
3.
1000.0001

Enter a name for the file.
numbers.txt

data[0] = 3.141590
data[1] = 9.990000
data[2] = 1.500000
data[3] = 3.000000
data[4] = 1000.000122
```

7.2 Formatted File Input

For formatted file input, use the fscanf() library function, which is used like scanf(), except that input comes from a specified stream instead of from stdin. The prototype for fscanf() is

```
int fscanf(FILE *fp, const char *fmt, ...);
```

The argument fp is the pointer to type FILE returned by fopen(), and fmt is a pointer to the format string that specifies how fscanf() is to read the input. The components of the format string are the same as for scanf(). Finally, the ellipses (...) indicate one or more additional arguments, the addresses of the variables where fscanf() is to assign the input.

The function fscanf() works exactly the same as scanf(), except that characters are taken from the specified stream rather than from stdin. To demonstrate fscanf(), you need a text file containing some numbers or strings in a format that can be read by the function.
Use your editor to create a file named INPUT.TXT, and enter five floating-point numbers with some space between them (spaces or newlines). For example, your file might look like this:

SPACE FOR HINT

```
123.45    87.001
100.02
0.00456   1.0005
```

Now, compile and run Listing 16.3.

Listing 16.3. Using fscanf() to read formatted data from a disk file.

```
1: /* Reading formatted file data with fscanf(). */
2: #include <stdlib.h>
3: #include <stdio.h>
4:
5: main()
6: {
7:    float f1, f2, f3, f4, f5;
8:    FILE *fp;
9:
10:   if ( (fp = fopen("INPUT.TXT", "r")) == NULL)
11:   {
12:      fprintf(stderr, "Error opening file.\n");
13:      exit(1);
14:   }
15:
16:   fscanf(fp, "%f %f %f %f %f", &f1, &f2, &f3, &f4, &f5);
17:   printf("The values are %f, %f, %f, %f, and %f\n.",
18:          f1, f2, f3, f4, f5);
19:
20:   fclose(fp);
21:    return(0);
22: }
```

The values are 123.45, 87.0001, 100.02, 0.00456, and 1.0005.

8. Character Input and Output

When used with disk files, the term character I/O refers to single characters as well as lines of characters. Remember, a line is a sequence of zero or more characters terminated by the newline character. Use character I/O with text-mode files. The following sections describe character input/output functions, and then you'll see a demonstration program.

8.1 Character Input

There are three character input functions: getc() and fgetc() for single characters, and fgets() for lines.

8.1.1 The getc() and fgetc() Functions

SPACE FOR HINT

The functions getc() and fgetc() are identical and can be used interchangeably. They input a single character from the specified stream. Here is the prototype of getc(), which is in STDIO.H:

int getc(FILE *fp);

The argument fp is the pointer returned by fopen() when the file is opened. The function returns the character that was input or EOF on error.

If getc() and fgetc() return a single character, why are they prototyped to return a type int? The reason is that, when reading files, you need to be able to read in the end-of-file marker, which on some systems isn't a type char but a type int.

8.1.2 The fgets() Function

To read a line of characters from a file, use the fgets() library function. The prototype is

char *fgets(char *str, int n, FILE *fp);

The argument str is a pointer to a buffer in which the input is to be stored, n is the maximum number of characters to be input, and fp is the pointer to type FILE that was returned by fopen() when the file was opened.

When called, fgets() reads characters from fp into memory, starting at the location pointed to by str. Characters are read until a newline is encountered or until n-1 characters have been read, whichever occurs first. By setting n equal to the number of bytes allocated for the buffer str, you prevent input from overwriting memory beyond allocated space. (The n-1 is to allow space for the terminating \0 that fgets() adds to the end of the string.) If successful, fgets() returns str. Two types of errors can occur, as indicated by the return value of NULL:

If a read error or EOF is encountered before any characters have been assigned to str, NULL is returned, and the memory pointed to by str is unchanged.

If a read error or EOF is encountered after one or more characters have been assigned to str, NULL is returned, and the memory pointed to by str contains garbage.

You can see that fgets() doesn't necessarily input an entire line (that is, everything up to the next newline character). If n-1 characters are read before a newline is encountered, fgets() stops. The next read operation from the file starts where the last one leaves off. To be sure that fgets() reads in entire strings, stopping only at newlines, be sure that the size of

your input buffer and the corresponding value of n passed to fgets() are large enough.

SPACE FOR HINT

8.2. Character Output

You need to know about two character output functions: putc() and fputs().

8.2.1 The putc() Function

The library function putc() writes a single character to a specified stream. Its prototype in STDIO.H reads

```
int putc(int ch, FILE *fp);
```

The argument ch is the character to output. As with other character functions, it is formally called a type int, but only the lower-order byte is used. The argument fp is the pointer associated with the file (the pointer returned by fopen() when the file was opened). The function putc() returns the character just written if successful or EOF if an error occurs. The symbolic constant EOF is defined in STDIO.H, and it has the value -1. Because no "real" character has that numeric value, EOF can be used as an error indicator (with text-mode files only).

8.2.2 The fputs() Function

To write a line of characters to a stream, use the library function fputs(). This function works just like puts(). The only difference is that with fputs() you can specify the output stream. Also, fputs() doesn't add a newline to the end of the string; if you want it, you must explicitly include it. Its prototype in STDIO.H is

```
char fputs(char *str, FILE *fp);
```

The argument str is a pointer to the null-terminated string to be written, and fp is the pointer to type FILE returned by fopen() when the file was opened. The string pointed to by str is written to the file, minus its terminating \0. The function fputs() returns a nonnegative value if successful or EOF on error.

9. Direct File Input and Output

You use direct file I/O most often when you save data to be read later by the same or a different C program. Direct I/O is used only with binary-mode files. With direct output, blocks of data are written from memory to disk. Direct input reverses the process: A block of data is read from a disk file into memory. For example, a single direct-output function call can write an entire array of type double to disk, and a single direct-input

SPACE FOR HINT

function call can read the entire array from disk back into memory. The direct I/O functions are fread() and fwrite().

9.1 The fwrite() Function

The fwrite() library function writes a block of data from memory to a binary-mode file. Its prototype in STDIO.H is

```
int fwrite(void *buf, int size, int count, FILE *fp);
```

The argument buf is a pointer to the region of memory holding the data to be written to the file. The pointer type is void; it can be a pointer to anything.

The argument size specifies the size, in bytes, of the individual data items, and count specifies the number of items to be written. For example, if you wanted to save a 100-element integer array, size would be 2 (because each int occupies 2 bytes) and count would be 100 (because the array contains 100 elements). To obtain the size argument, you can use the sizeof() operator.

The argument fp is, of course, the pointer to type FILE, returned by fopen() when the file was opened. The fwrite() function returns the number of items written on success; if the value returned is less than count, it means that an error has occurred. To check for errors, you usually program fwrite() as follows:

```
if( (fwrite(buf, size, count, fp)) != count)
fprintf(stderr, "Error writing to file.");
```

Here are some examples of using fwrite(). To write a single type double variable x to a file, use the following:

```
fwrite(&x, sizeof(double), 1, fp);
```

To write an array data[] of 50 structures of type address to a file, you have two choices:

```
fwrite(data, sizeof(address), 50, fp);
fwrite(data, sizeof(data), 1, fp);
```

The first method writes the array as 50 elements, with each element having the size of a single type address structure. The second method treats the array as a single element. The two methods accomplish exactly the same thing.

The following section explains fread() and then presents a program demonstrating fread() and fwrite().

The fread() library function reads a block of data from a binary-mode file into memory. Its prototype in STDIO.H is

int fread(void *buf, int size, int count, FILE *fp);

The argument buf is a pointer to the region of memory that receives the data read from the file. As with fwrite(), the pointer type is void.

The argument size specifies the size, in bytes, of the individual data items being read, and count specifies the number of items to read. Note how these arguments parallel the arguments used by fwrite(). Again, the sizeof() operator is typically used to provide the size argu-ment. The argument fp is (as always) the pointer to type FILE that was returned by fopen() when the file was opened. The fread() function returns the number of items read; this can be less than count if end-of-file was reached or an error occurred.

SPACE FOR HINT

Listing : Using fwrite() and fread() for direct file access.

```
1: /* Direct file I/O with fwrite() and fread(). */
2: #include <stdlib.h>
3: #include <stdio.h>
4:
5: #define SIZE 20
6:
7: main()
8: {
9:    int count, array1[SIZE], array2[SIZE];
10:   FILE *fp;
11:
12:   /* Initialize array1[]. */
13:
14:   for (count = 0; count < SIZE; count++)
15:      array1[count] = 2 * count;
16:
17:   /* Open a binary mode file. */
18:
19:   if ( (fp = fopen("direct.txt", "wb")) == NULL)
20:   {
21:      fprintf(stderr, "Error opening file.");
22:      exit(1);
23:   }
24:   /* Save array1[] to the file. */
25:
26:   if (fwrite(array1, sizeof(int), SIZE, fp) != SIZE)
27:   {
28:      fprintf(stderr, "Error writing to file.");
29:      exit(1);
30:   }
```

SPACE FOR HINT

```
31:
32:    fclose(fp);
33:
34:    /* Now open the same file for reading in binary mode.
35:
36:    if ( (fp = fopen("direct.txt", "rb")) == NULL)
37:    {
38:       fprintf(stderr, "Error opening file.");
39:       exit(1);
40:    }
41:
42:    /* Read the data into array2[]. */
43:
44:    if (fread(array2, sizeof(int), SIZE, fp) != SIZE)
45:    {
46:       fprintf(stderr, "Error reading file.");
47:       exit(1);
48:    }
49:
50:    fclose(fp);
51:
52:    /* Now display both arrays to show they're the same.
53:
54:    for (count = 0; count < SIZE; count++)
55:       printf("%d\t%d\n", array1[count], array2[count]);
56:    return(0);
57: }
0       0
2       2
4       4
6       6
8       8
10      10
12      12
14      14
16      16
18      18
20      20
22      22
24      24
26      26
28      28
30      30
32      32
34      34
36      36
38      38
```

SPACE FOR HINT

Every open file has a file position indicator associated with it. The position indicator specifies where read and write operations take place in the file. The position is always given in terms of bytes from the beginning of the file. When a new file is opened, the position indicator is always at the beginning of the file, position 0. (Because the file is new and has a length of 0, there's no other location to indicate.) When an existing file is opened, the position indicator is at the end of the file if the file was opened in append mode, or at the beginning of the file if the file was opened in any other mode.

Writing and reading operations occur at the location of the position indicator and update the position indicator as well.

For example, if you open a file for reading, and 10 bytes are read, you input the first 10 bytes in the file (the bytes at positions 0 through 9). After the read operation, the position indicator is at position 10, and the next read operation begins there. Thus, if you want to read all the data in a file sequentially or write data to a file sequentially, you don't need to be concerned about the position indicator, because the stream I/O functions take care of it automatically.

When you need more control, use the C library functions that let you determine and change the value of the file position indicator. By controlling the position indicator, you can per-form random file access. Here, random means that you can read data from, or write data to, any position in a file without reading or writing all the preceding data.

10.1 The ftell() and rewind() Functions

To set the position indicator to the beginning of the file, use the library function rewind(). Its prototype, in STDIO.H, is

```
void rewind(FILE *fp);
```

The argument fp is the FILE pointer associated with the stream. After rewind() is called, the file's position indicator is set to the beginning of the file (byte 0). Use rewind() if you've read some data from a file and you want to start reading from the beginning of the file again without closing and reopening the file.

To determine the value of a file's position indicator, use ftell(). This function's prototype, located in STDIO.H, reads

```
long ftell(FILE *fp);
```

The argument fp is the FILE pointer returned by fopen() when the file was opened. The function ftell() returns a type long that gives the current file position in bytes from the start of the file (the first byte is at position 0). If an error occurs, ftell() returns -1L (a type long -1).

SPACE FOR HINT

Listing : Using ftell() and rewind().

```
1:  /* Demonstrates ftell() and rewind(). */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  #define BUFLEN 6
6:
7:  char msg[] = "abcdefghijklmnopqrstuvwxyz";
8:
9:  main()
10: {
11:    FILE *fp;
12:    char buf[BUFLEN];
13:
14:    if ( (fp = fopen("TEXT.TXT", "w")) == NULL)
15:    {
16:       fprintf(stderr, "Error opening file.");
17:       exit(1);
18:    }
19:
20:    if (fputs(msg, fp) == EOF)
21:    {
22:       fprintf(stderr, "Error writing to file.");
23:       exit(1);
24:    }
25:
26:    fclose(fp);
27:
28:    /* Now open the file for reading. */
29:
30:    if ( (fp = fopen("TEXT.TXT", "r")) == NULL)
31:    {
32:       fprintf(stderr, "Error opening file.");
33:       exit(1);
34:    }
35:    printf("\nImmediately after opening, position = %ld", ftell(fp));
36:
37:    /* Read in 5 characters. */
38:
39:    fgets(buf, BUFLEN, fp);
40:    printf("\nAfter reading in %s, position = %ld", buf, ftell(fp));
41:
42:    /* Read in the next 5 characters. */
43:
44:    fgets(buf, BUFLEN, fp);
45:       printf("\n\nThe next 5 characters are %s, and position now =
%ld",
```

```
47:
48:    /* Rewind the stream. */
49:
50:    rewind(fp);
51:
52:    printf("\n\nAfter rewinding, the position is back at %ld",
53:           ftell(fp));
54:
55:    /* Read in 5 characters. */
56:
57:    fgets(buf, BUFLEN, fp);
58:    printf("\nand reading starts at the beginning again: %s\n", buf);
59:    fclose(fp);
60:    return(0);
61: }
```

SPACE FOR HINT

```
Immediately after opening, position = 0
After reading in abcde, position = 5
The next 5 characters are fghij, and position now = 10
After rewinding, the position is back at 0
and reading starts at the beginning again: abcde
```

10.2 The fseek() Function

More precise control over a stream's position indicator is possible with the fseek() library function. By using fseek(), you can set the position indicator anywhere in the file. The function prototype, in STDIO.H, is

```
int fseek(FILE *fp, long offset, int origin);
```

The argument fp is the FILE pointer associated with the file. The distance that the position indicator is to be moved is given by offset in bytes. The argument origin specifies the move's relative starting point. There can be three values.

Table of Possible origin values for fseek().

Constant	Value	Description
SEEK_SET	0	Moves the indicator offset bytes from the beginning of the file.
SEEK_CUR	1	Moves the indicator offset bytes from its current position.
SEEK_END	2	Moves the indicator offset bytes from the end of the file.

The function fseek() returns 0 if the indicator was successfully moved or nonzero if an error occurred.

Listing : Random file access with fseek().

SPACE FOR HINT

```
1:  /* Random access with fseek(). */
2:
3:  #include <stdlib.h>
4:  #include <stdio.h>
5:
6:  #define MAX 50
7:
8:  main()
9:  {
10:    FILE *fp;
11:    int data, count, array[MAX];
12:    long offset;
13:
14:    /* Initialize the array. */
15:
16:    for (count = 0; count < MAX; count++)
17:       array[count] = count * 10;
18:
19:    /* Open a binary file for writing. */
20:
21:    if ( (fp = fopen("RANDOM.DAT", "wb")) == NULL)
22:    {
23:       fprintf(stderr, "\nError opening file.");
24:       exit(1);
25:    }
26:
27:    /* Write the array to the file, then close it. */
28:
29:    if ( (fwrite(array, sizeof(int), MAX, fp)) != MAX)
30:    {
31:       fprintf(stderr, "\nError writing data to file.");
32:       exit(1);
33:    }
34:
35:    fclose(fp);
36:
37:    /* Open the file for reading. */
38:
39:    if ( (fp = fopen("RANDOM.DAT", "rb")) == NULL)
40:    {
41:       fprintf(stderr, "\nError opening file.");
42:       exit(1);
43:    }
44:
45:    /* Ask user which element to read. Input the element */
46:    /* and display it, quitting when -1 is entered. */
47:
48:    while (1)
49:    {
```

```
50:       printf("\nEnter element to read, 0-%d, -1 to quit: ",MAX-1);
51:       scanf("%ld", &offset);
52:
53:       if (offset < 0)
54:          break;
55:       else if (offset > MAX-1)
56:          continue;
57:
58:       /* Move the position indicator to the specified element. */
59:
60:       if ( (fseek(fp, (offset*sizeof(int)), SEEK_SET)) != 0)
61:       {
62:          fprintf(stderr, "\nError using fseek().");
63:          exit(1);
64:       }
65:
66:       /* Read in a single integer. */
67:
68:       fread(&data, sizeof(int), 1, fp);
69:
70:       printf("\nElement %ld has value %d.", offset, data);
71:    }
72:
73:    fclose(fp);
74:    return(0);
75: }
```

```
Enter element to read, 0-49, -1 to quit: 5
Element 5 has value 50.
Enter element to read, 0-49, -1 to quit: 6
Element 6 has value 60.
Enter element to read, 0-49, -1 to quit: 49
Element 49 has value 490.
Enter element to read, 0-49, -1 to quit: 1
Element 1 has value 10.
Enter element to read, 0-49, -1 to quit: 0
Element 0 has value 0.
Enter element to read, 0-49, -1 to quit: -1
```

10.3 Detecting the End of a File

Sometimes you know exactly how long a file is, so there's no need to be able to detect the file's end. For example, if you used fwrite() to save a 100-element integer array, you know the file is 200 bytes long (assuming 2-byte integers). At other times, however, you don't know how long the file is, but you still want to read data from the file, starting at the beginning and proceeding to the end. There are two ways to detect end-of-file.

SPACE FOR HINT

SPACE FOR HINT

When reading from a text-mode file character-by-character, you can look for the end-of-file character. The symbolic constant EOF is defined in STDIO.H as -1, a value never used by a "real" character. When a character input function reads EOF from a text-mode stream, you can be sure that you've reached the end of the file. For example, you could write the following:

```
while ( (c = fgetc( fp )) != EOF )
```

With a binary-mode stream, you can't detect the end-of-file by looking for -1, because a byte of data from a binary stream could have that value, which would result in premature end of input. Instead, you can use the library function feof(), which can be used for both binary- and text-mode files:

```
int feof(FILE *fp);
```

The argument fp is the FILE pointer returned by fopen() when the file was opened. The function feof() returns 0 if the end of file fp hasn't been reached, or a nonzero value if end-of-file has been reached. If a call to feof() detects end-of-file, no further read operations are permitted until a rewind() has been done, fseek() is called, or the file is closed and reopened.

The following program demonstrates the use of feof(). When you're prompted for a filename, enter the name of any text file--one of your C source files, for example, or a header file such as STDIO.H. Just be sure that the file is in the current directory, or else enter a path as part of the filename. The program reads the file one line at a time, displaying each line on stdout, until feof() detects end-of-file.

Listing : Using feof() to detect the end of a file.

```
1:  /* Detecting end-of-file. */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  #define BUFSIZE 100
6:
7:  main()
8:  {
9:      char buf[BUFSIZE];
10:     char filename[60];
11:     FILE *fp;
12:
13:     puts("Enter name of text file to display: ");
14:     gets(filename);
15:
16:     /* Open the file for reading. */
17:     if ( (fp = fopen(filename, "r")) == NULL)
```

```
18:    {
19:       fprintf(stderr, "Error opening file.");
20:       exit(1);
21:    }
22:
23:    /* If end of file not reached, read a line and display it. */
24:
25:    while ( !feof(fp) )
26:    {
27:       fgets(buf, BUFSIZE, fp);
28:       printf("%s",buf);
29:    }
30:
31:    fclose(fp);
32:    return(0);
33: }
Enter name of text file to display:
hello.c
#include <stdio.h>
main()
{
   printf("Hello, world.");
   return(0);
}
```

SPACE FOR HINT

10.4 Copying a File

It's frequently necessary to make a copy of a file--an exact duplicate with a different name (or with the same name but in a different drive or directory). In DOS, you do this with the COPY command, and other operating systems have equivalents.

How do you copy a file in C? There's no library function, so you need to write your own.

This might sound a bit complicated, but it's really quite simple thanks to C's use of streams for input and output. Here are the steps you follow:

1. Open the source file for reading in binary mode. (Using binary mode ensures that the function can copy all sorts of files, not just text files.)

2. Open the destination file for writing in binary mode.

3. Read a character from the source file. Remember, when a file is first opened, the pointer is at the start of the file, so there's no need to position the file pointer explicitly.

4. If the function feof() indicates that you've reached the end of the source file, you're done and can close both files and return to the calling program.

SPACE FOR HINT

5. If you haven't reached end-of-file, write the character to the destination file, and then loop back to step 3.

The following program contains a function, copy_file(), that is passed the names of the source and destination files and then performs the copy operation just as the preceding steps outlined. If there's an error opening either file, the function doesn't attempt the copy operation and returns -1 to the calling program. When the copy operation is complete, the program closes both files and returns 0.

A function that copies a file.

```
1:  /* Copying a file. */
2:
3:  #include <stdio.h>
4:
5:  int file_copy( char *oldname, char *newname );
6:
7:  main()
8:  {
9:      char source[80], destination[80];
10:
11:     /* Get the source and destination names. */
12:
13:     printf("\nEnter source file: ");
14:     gets(source);
15:     printf("\nEnter destination file: ");
16:     gets(destination);
17:
18:     if ( file_copy( source, destination ) == 0 )
19:        puts("Copy operation successful");
20:     else
21:        fprintf(stderr, "Error during copy operation");
22:     return(0);
23: }
24: int file_copy( char *oldname, char *newname )
25: {
26:     FILE *fold, *fnew;
27:     int c;
28:
29:     /* Open the source file for reading in binary mode. */
30:
31:     if ( ( fold = fopen( oldname, "rb" ) ) == NULL )
32:        return -1;
33:
34:     /* Open the destination file for writing in binary mode. */
35:
36:     if ( ( fnew = fopen( newname, "wb" ) ) == NULL )
```

```
37:    {
38:       fclose ( fold );
39:       return -1;
40:    }
41:
42:    /* Read one byte at a time from the source; if end of file */
43:    /* has not been reached, write the byte to the */
44:    /* destination. */
45:
46:    while (1)
47:    {
48:       c = fgetc( fold );
49:
50:       if ( !feof( fold ) )
51:          fputc( c, fnew );
52:       else
53:          break;
54:    }
55:
56:    fclose ( fnew );
57:    fclose ( fold );
58:
59:    return 0;
60: }
Enter source file: list1610.c
Enter destination file: tmpfile.c
Copy operation successful
```

SPACE FOR HINT

11. Command-line Arguments

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When main is called, it is called with two arguments.

The first (conventionally called argc, for argument count) is the number of command-line arguments the program was invoked with; the second (argv, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

The simplest illustration is the program echo, which echoes its command-line arguments on a single line, separated by blanks. That is, the command

```
echo hello, world
```

prints the output

```
hello, world
```

SPACE FOR HINT

By convention, argv[0] is the name by which the program was invoked, so argc is at least 1. If argc is 1, there are no command-line arguments after the program name.

The first version of echo treats argv as an array of character pointers:

```
#include <stdio.h>
/* echo command-line arguments; 1st version */

main(int argc, char *argv[])
{
   int i;
   for (i = 1; i < argc; i++)
      printf("%s%s", argv[i], (i < argc-1) ? " " : "");
   printf("\n");
   return 0;
}
```

Since argv is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. This next variant is based on incrementing argv, which is a pointer to pointer to char, while argc is counted down:

```
#include <stdio.h>
/* echo command-line arguments; 2nd version */
main(int argc, char *argv[])
{
   while (--argc > 0)
      printf("%s%s", *++argv, (argc > 1) ? " " : "");
   printf("\n");
   return 0;
}
```

Since argv is a pointer to the beginning of the array of argument strings, incrementing it by 1 (++argv) makes it point at the original argv[1] instead of argv[0]. Each successive increment moves it along to the next argument; *argv is then the pointer to that argument. At the same time, argc is decremented; when it becomes zero, there are no arguments left to print.

Alternatively, we could write the printf statement as

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

This shows that the format argument of printf can be an expression too.

C.Y.P

1. What is File ?

2. Explain about Opening and Closing of Data Files.

3. Explain the steps to create a Data File.

4. Explain about Unformatted Data File.

5. Discuss about Command Line argument.

LIST OF EXAMPLE PROGRAM

1. LARGEST AMONG THREE NUMBERS

SPACE FOR HINT

```
#include<stdio.h>
#include<conio.h>
main()
{
int a,b,c;
printf("Enter A,B,C values\n");
scanf("%d %d %d",&a,&b,&c);
if((a>b)&&(a>c))
printf("A is greater than B and C");
else
if((b>a)&&(b>c))
printf("B is greater than A and C");
else
printf("C is greather than B and A");
getch();
}
```

Input:
Enter A,B,C Values
8
7
9

Output:
C is greater than B and A

2. QUADRATIC EQUATION CALCULATION

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{
float a,b,c,d,r1,r2;
clrscr();
printf("Enter the number\n");
scanf("%f%f%f",&a,&b,&c);
d=pow(b,2)-4*a*c;
if(d>0)
{
printf("Real & +ve roots");
r1=(-b+sqrt(d))/(2.0*a);
r2=(-b-sqrt(d))/(2.0*a);
printf("r1&r2 are %f\t%f",r1,r2);
}
else if(d==0)
{
printf("Equal roots");
```

SPACE FOR HINT

```
r1=-b/(2.0*a);
r2=r1;
printf("r1&r2 are %f\t%f",r1,r2);
}
else
{
printf("Complex roots");
d=-d;
r1=sqrt(d)/(2.0*a);
r2=-b/(2.0*a);
printf("r1&r2 are %f\t%f",r1,r2);
}
}
```

Input:
Enter the number
1 -5 6
Output:
Real & +ve roots r1 &r2 are 3.000000 2.000000

3. ASCENDING ORDER OF NUMBER USING ARRAY

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,j,a[10],t,n;
clrscr();
printf("Enter the value:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
for(i=0;i<n;i++)
{
for(j=i+1;j<n;j++)
{
if(a[i]>a[j])
{
t=a[i];
a[i]=a[j];
a[j]=t;
}
}
}
printf("\n");
printf("The Asending order is \n");
for(i=0;i<n;i++)
printf("%d\n",a[i]);
```

```
getch();
}
```

SPACE FOR HINT

Input:
Enter the value:5
4
24
15
64
8
Output:
4
8
15
24
64

4. IMPLEMETATION OF LINEAR SEARCH

```
#include<stdio.h>
#include<conio.h>
main()
{
int s,n,a[50],i,f=0;
clrscr();
printf("\n Enter the values for N terms:");
scanf("%d",&n);
printf("\n Enter the values for the Array:");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("\n Enter the number:");
scanf("%d",&s);
for(i=0;i<n;i++)
{
if(s==a[i])
f=1;
}
if(f==0)
printf("The given number is not present");
else
printf("The given number is present");
getch();
}
```

Input:
Enter the values for N terms:3

Enter the values for the Array:

SPACE FOR HINT

2
4
6

Enter the number:4

Output:
The given number is present

5. MATRIX ADDITION USING ARRAYS

```
#include<stdio.h>
#include<conio.h>
main()
{
int a[10][10],b[10][10],c[10][10],i,j,k,r1,r2,c1,c2;
clrscr();
printf("Enter the order for the matrix A\n");
scanf("%d",&r1);
scanf("%d",&c1);
printf("Enter the order for the matrix B");
scanf("%d",&r2);
scanf("%d",&c2);
printf("Elements of A \n");
for(i=0;i<r1;i++)
for(j=0;j<c1;j++)
scanf("%d",&a[i][j]);
printf("The matrix A is :\n");
for(i=0;i<r1;i++)
{
for(j=0;j<c1;j++)
{
printf("%d\t",a[i][j]);
}
printf("\n");
}
printf("Elements of B:");
for(i=0;i<r2;i++)
for(j=0;j<c2;j++)
scanf("%d",&b[i][j]);
printf("The matrix B is :\n");
for(i=0;i<r2i++)
{
for(j=0;j<c2;j++)
{
printf("%d\t",b[i][j]);
}
printf("\n");
```

```
for(j=0;j<c1;j++)
{
c[i][j]=a[i][j]+b[i][j];
}
printf("Output\n");
{
for(i=0;i<r1;i++)
{
for(j=0;j<c1;j++)
printf("%d\t",c[i][j]);
printf("\n");
}
getch();
}
}
```

SPACE FOR HINT

Input:
Enter the order for the matrix A
2
2

Enter the order for the matrix B
2
2
Elements of A :
1
2
2
3
The matrix A is :
1 2
2 3

Elements of B :
1
2
3
4

The matrix B is :
1 2
3 4

Output:
2 4
5 7

MATRIX SUBTRACTION USING ARRAYS

SPACE FOR HINT

```
#include<stdio.h>
#include<conio.h>
main()
{
int a[10][10],b[10][10],c[10][10],i,j,k,r1,r2,c1,c2;
clrscr();
printf("Enter the order for the matrix A\n");
scanf("%d",&r1);
scanf("%d",&c1);
printf("Enter the order for the matrix B");
scanf("%d",&r2);
scanf("%d",&c2);
printf("Elements of A \n");
for(i=0;i<r1;i++)
for(j=0;j<c1;j++)
scanf("%d",&a[i][j]);
printf("The matrix A is :\n");
for(i=0;i<r1;i++)
{
for(j=0;j<c1;j++)
{
printf("%d\t",a[i][j]);
}
printf("\n");
}
printf("Elements of B:");
for(i=0;i<r2;i++)
for(j=0;j<c2;j++)
scanf("%d",&b[i][j]);
printf("The matrix B is :\n");
for(i=0;i<r2;i++)
{
for(j=0;j<c2;j++)
{
printf("%d\t",b[i][j]);
}
printf("\n");
}
for(i=0;i<r1;i++)
for(j=0;j<c1;j++)
{
c[i][j]=a[i][j]-b[i][j];
}
printf("Output\n");
{
for(i=0;i<r1;i++)
{
```

```
printf("\n");
}
getch();
}
}
```

Input:
Enter the order for the matrix A:
2
2

Enter the order for the matrix B:
2
2
Elements of A:
2
4
6
8
The matrix A
2 4
6 8

Elements of B:
1
2
3
4

The matrix B is :
1 2
3 4

Output:
1 2
3 4

SPACE FOR HINT

7. MATRIX MULTIPLICATION USING ARRAYS

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{
int a[3][3],b[3][3],c[3][3],i,j,k,r1,r2,c1,c2;
clrscr();
printf("Enter the order of A matrix\n");
scanf("%d%d",&r1,&c1);
```

SPACE FOR HINT

```
printf("Enter the order of B matrix\n");
scanf("%d%d",&r2,&c2);
if(c1!=r2)
printf("Multiplication not perform");
else
{
printf("Enter A matrix \n");
{
for(i=0;i<r1;i++)
{
for(j=0;j<c1;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("Enter B matrix");
for(i=0;i<r2;i++)
{
for(j=0;j<c2;j++)
{
scanf("%d",&b[i][j]);
}
}
for(i=0;i<r1;i++)
{
for(j=0;j<c2;j++)
{
c[i][j]=0;
for(k=0;k<c1;k++)
{
c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
}
}
printf("Multiplication of two matrix \n");
for(i=0;i<r1;i++)
{
for(j=0;j<c2;j++)
{
printf("%d\t",c[i][j]);
}
printf("\n");
}
getch();
}
}
}
```

Input:

SPACE FOR HINT

Enter the order of A matrix
2
2

Enter the order of B matrix
2
2

Enter A matrix
1
2
3
4

Enter B matrix
1
2
3
4

Output:
Multiplication of two matrix
7 10
15 22

8. SORTING THE NAME USING ARRAY

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char name[5][20],temp[20];
int i,j,n;
clrscr();
printf("Enter n names:");
scanf("%d",&n);
printf("Enter the names \n");
for(i=0;i<=n;i++)
gets(name[i]);
for(i=0;i<n-1;i++)
{
for(j=i+1;j<n;j++)
{
if(strcmp(name[i],name[j])>0)
{
strcpy(temp,name[i]);
strcpy(name[i],name[j]);
strcpy(name[j],temp);
```

SPACE FOR HINT

```
}
}
}
printf("The sorted names are:");
for(i=0;i<=n;i++)
puts(name[i]);
getch();
}
```

Input:
Enter n names:4
Enter the names
manoj
anitha
tyson
suresh

Output:

The sorted names are:
anitha
manoj
suresh
tyson

9. FUNCTIONS USING CALL BY VALUE

```
#include<stdio.h>
#include<conio.h>
main()
{
int a,b;
clrscr();
void swap(int,int);
printf("Enter two numbers\n");
scanf("%d %d",&a,&b);
printf("\n Values before swap a=%d b=%d",a,b);
swap(a,b);
printf("\n Values after swap a=%d b=%d",a,b);
getch();
}
void swap(int x,int y)
{
int temp;
temp=x;
x=y;
y=temp;
}
```

Input:
Enter two numbers
23
12
values before swap a=23 b=12
values after swap a=12 b=23

SPACE FOR HINT

10. FACTORIAL USING RECURSION

```
#include<stdio.h>
#include<conio.h>
main()
{
int n,f;
int fact(int);
printf("Enter n value\n");
scanf("%d",&n);
f=fact(n);
printf("Factorial value=%d",f);
getch();
}
int fact(int x)
{
int f1;
if(x==1)
return(1);
else
f1=x*fact(x-1);
return(f1);
}
```

Input:
Enter n values
3

Output:
Factorial value=6

11. PALINDROME CHECKING

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char str[80];
int i,j,len=0,flag=1;
clrscr();
printf("Enter the string:");
```

SPACE FOR HINT

```
scanf("%s",str);
len=strlen(str);
printf("Length of entered string=%d",len);
for(i=0,j=len-1;i<len/2;i++,j--)
{
if(str[i]!=str[j])
{
flag=0;
break;
}
}
if(flag)
printf("\n %s is palindrome",str);
else
printf("\n %s is not palindrome",str);
getch();
}
```

Input:
Enter the string: malayalam

Output:
Length of entered string=9
malayalam is palindrome

12. MULTIPLICATION TABLE

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,s,t;
clrscr();
printf("enter table number:\n");
scanf("%d",&t);
for(i=0;i<=10;i++)
{
s=i*t;
printf("%d*%d=%d\n",i,t,s);
}
getch();
}
```

Input:
enter table number:
5
Output:
0*5=0
1*5=5

SPACE FOR HINT

```
2*5=10
3*5=15
4*5=20
5*5=25
6*5=30
7*5=35
8*5=40
9*5=45
10*5=50
```

13. CALCULATION OF SIMPLE INTEREST

```
#include<stdio.h>
#include<conio.h>
main()
{
float p,n,r,i;
clrscr();
printf("Enter the principal value \n");
scanf("%f",&p);
printf("Enter the number of years \n");
scanf("%f",&n);
printf("Enter the rate of interest \n");
scanf("%f",&r);
i=(p*n*r)/100;
printf("simple interest %f",i);
getch();
}
```

Input:
Enter the principal value
50000
Enter the number of years
2
Enter the rate of interest
1.5

Output:
simple interest 1500.000000

14. FINDING OUT THE SUM OF DIGITS

```
#include<stdio.h>
#include<conio.h>
main()
{
int n,r,s;
```

SPACE FOR HINT

```
s=0;
clrscr();
printf("\n Enter the value");
scanf("%d",&n);
while(n>0)
{
r=n%10;
s=s+r;
n=n/10;
}
printf("\n sum of the digits:%d",s);
getch();
}
```

Input:

Enter the value
234

Output:

Sum of the digits : 9

15. REVERSE OF THE GIVEN DIGIT

```
#include<stdio.h>
#include<conio.h>
main()
{
int n,r,q;
clrscr();
printf("Enter the number:\n");
scanf("%d",&n);
while(n>0)
{
r=n%10;
printf("%d",r);
n=n/10;
}
getch();
}
```

Input:

Enter the number :
123

Output:
321

16. CHECK THE GIVEN NUMBER IS PRIME NUMBER OR NOT

SPACE FOR HINT

```
#include<stdio.h>
#include<conio.h>
main()
{
int num,i;
clrscr();
printf("Enter the number \n");
scanf("%d",&num);
i=2;
while(i<=num-1)
{
if(num%i==0)
{
printf("It's not a prime number");
break;
}
i++;
}
if(i==num)
printf("It's a prime number");
getch();
}
```

Input:
Enter the number
15

Output :
It's not a prime number

Input:
Enter the number
11

Output:
It's a prime number

17. PRIME NUMBER GENERATION

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,j,n,f;
clrscr();
printf("Enter the number \n");
```

SPACE FOR HINT

```
scanf("%d",&n);
for(i=3;i<=n;i++)
{
f=0;
for(j=2;j<i;j++)
{
if(i%j==0)
f=1;
}
if(f==0)
printf("%d \n",j);
}
getch();
}
```

Input:
Enter the number
12

Output:
3
5
7
11

18. GENERATING FIBONACCI SERIES

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,f1=0,f2=1,f,n;
clrscr();
printf("Enter the number\n");
scanf("%d",&n);
printf("%d\n",f1);
printf("%d\n",f2);
for(i=3;i<=n;i++)
{
f=f1+f2;
printf("%d \n",f);
f1=f2;
f2=f;
}
getch();
}
```

Input:

Enter the number
7

Output:
0
1
1
2
3
5
8

SPACE FOR HINT

19. ARMSTRONG NUMBER GENERATION

```
#include<stdio.h>
#include<conio.h>
main()
{
int n,b,s,t,a,i;
clrscr();
printf("Enter the number:\n");
scanf("%d",&n);
for(i=0;i<=n;i++)
{
a=i;
s=0;
while(a>0)
{
b=a%10;
s=s+(b*b*b);
a=a/10;
}
if(s==i)
printf("\n %d",i);
}
getch();
}
```

Input:
Enter the number:
500

Output:
1
153
370
371
407

SPACE FOR HINT

20. a) FILE CONCEPT : WRITING A FILE

```
#include<stdio.h>
#include<conio.h>
main()
{
FILE *fp1;
char c;
clrscr();
fp1=fopen("abc.dat","w");
printf("Enter message to exit press return key\n");
while((c=getchar())!='\n')
putc(c,fp1);
fclose(fp1);
getch();
}
```

Output:
Enter message to exit press return key
Welcome to c lab

20. b) FILE CONCEPT :READING A FILE

```
#include<stdio.h>
#include<conio.h>
main()
{
FILE *fp1;
char ch;
int noa,noe,noi,noo,nou;
noa=noe=noi=noo=nou=0;
fp1=fopen("abc.dat","r");
clrscr();
while((ch=getc(fp1))>0)
{
if(ch=='a')noa++;
if(ch=='e')noe++;
if(ch=='i')noi++;
if(ch=='o')noo++;
if(ch=='u')nou++;
}
printf("\nNumber of a %d",noa);
printf("\nNumber of e %d",noe);
printf("\nNumber of i %d",noi);
printf("\nNumber of o %d",noo);
printf("\nNumber of u %d",nou);
fclose(fp1);
getch();
}
```

TABLE OF CONTENTS

SPACE FOR HINT

UNIT - 6

6. Introduction : Principles of Object Oriented Programming

OOP is a design philosophy. It stands for Object Oriented Programming. **O**bject-**O**riented **P**rogramming (*OOP*) uses a different set of programming languages than old procedural programming languages (*C, Pascal*, etc.). Everything in *OOP* is grouped as self sustainable "*objects*".

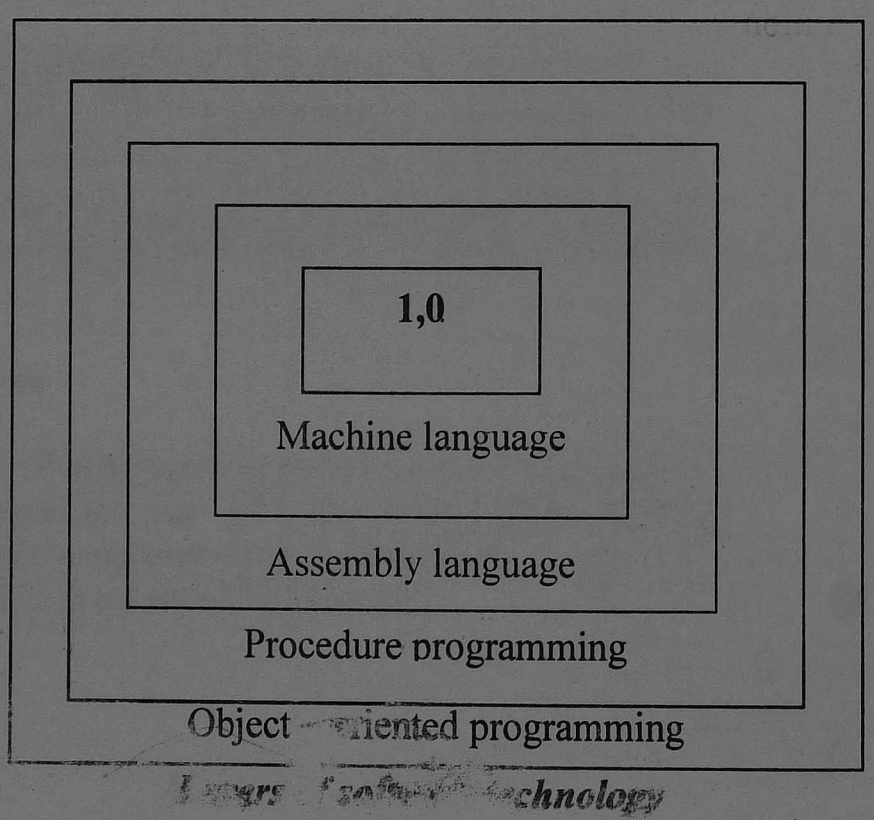
The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages. The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods.

While designing C++ modules, we try to see whole world in the form of objects.

For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

6.1 Software evolution

Ernest tello, a well-known writer in field of artificial intelligence, compared to evolution of software technology to the growth of a tree. Like a tree, the software evolution has had distinct phases or "layers" of growth. These layers were built up one by one over the last four decades as shown below



SPACE FOR HINT

Each layer representing an improvement over the previous one.

Since the invention of the computer, many programming approaches have been tried. These included techniques such as modular programming, top-down programming, bottom-up programming and structure programming.

With the advent of language such as C, structured programming became very popular and was the main technique of the 1980s. Structured programming was powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the program grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

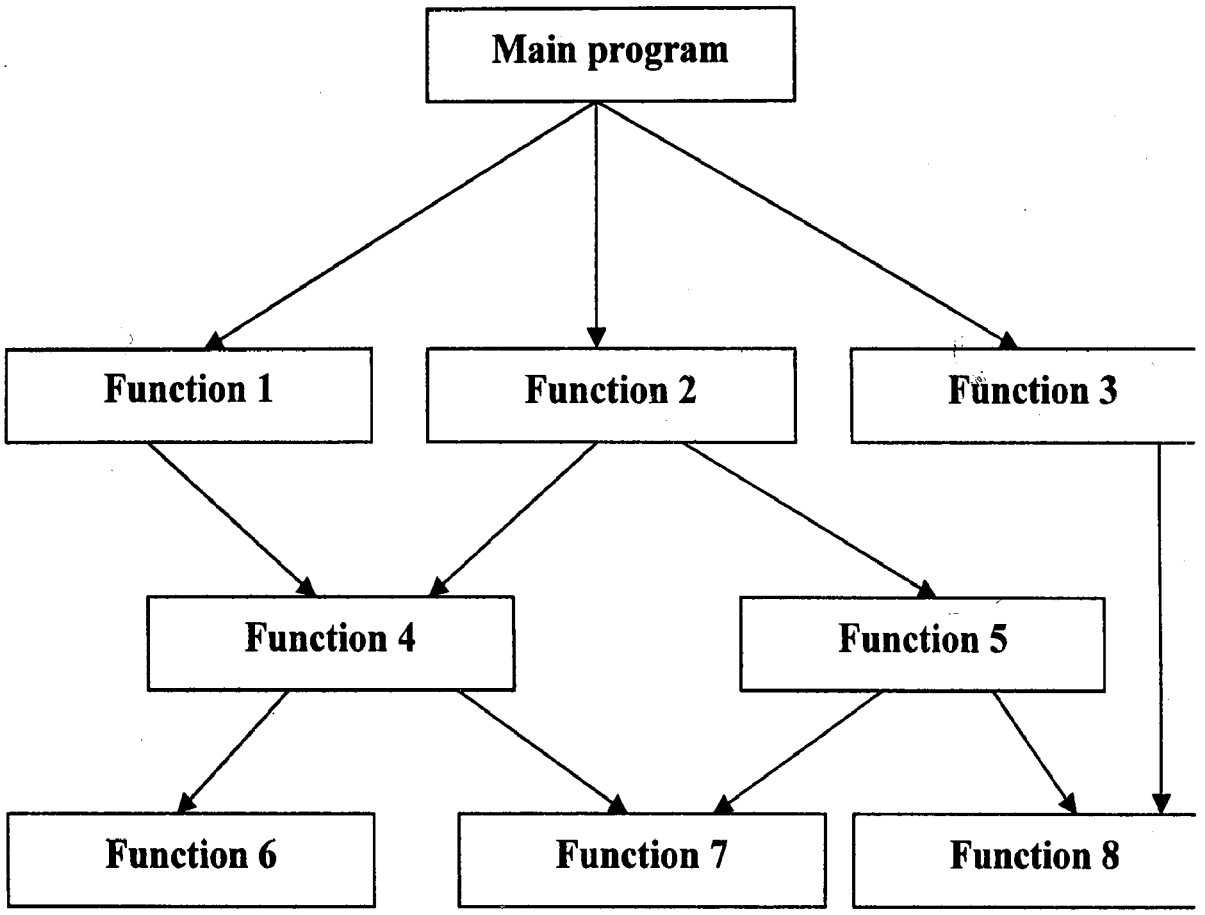
Object-Oriented Programming(OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing program and has nothing to do with any particular language. However, not all languages are suitable to implement of OOP concepts easily.

A LOOK AT PROCEDURE ORIENTED PROGRAMMING

Conventional programming using high level languages such as COBOL, FORTRAN and C is commonly known as procedure-oriented programming. In the procedure –oriented approach, the problem is viewed as a sequence of things to be done, such as reading, calculating and printing. A number of functions are written to accomplish these tasks. . The primary focus is on function. A typical program structure for procedural programming is **given below.** The technique of hierarchical decomposition has been used to specify the tasks to be completed in order to solve a problem.

SPACE FOR HINT

Typical structure of procedure-oriented programs



Procedure-oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instruction into groups known as functions. We normally use a flowchart to organize these actions and represent the flow of control from one action to another.

While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data ? How are they affected by the function that work on them ?

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data.

Global data are move vulnerable to an inadvertent change by function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we should also revise all functions that access the data.

Some characteristics exhibited by procedure-oriented programming are :

Emphasis is on doing things (algo.)

- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Function transforms data from one from to another.
- Employs top-down approach in program design.

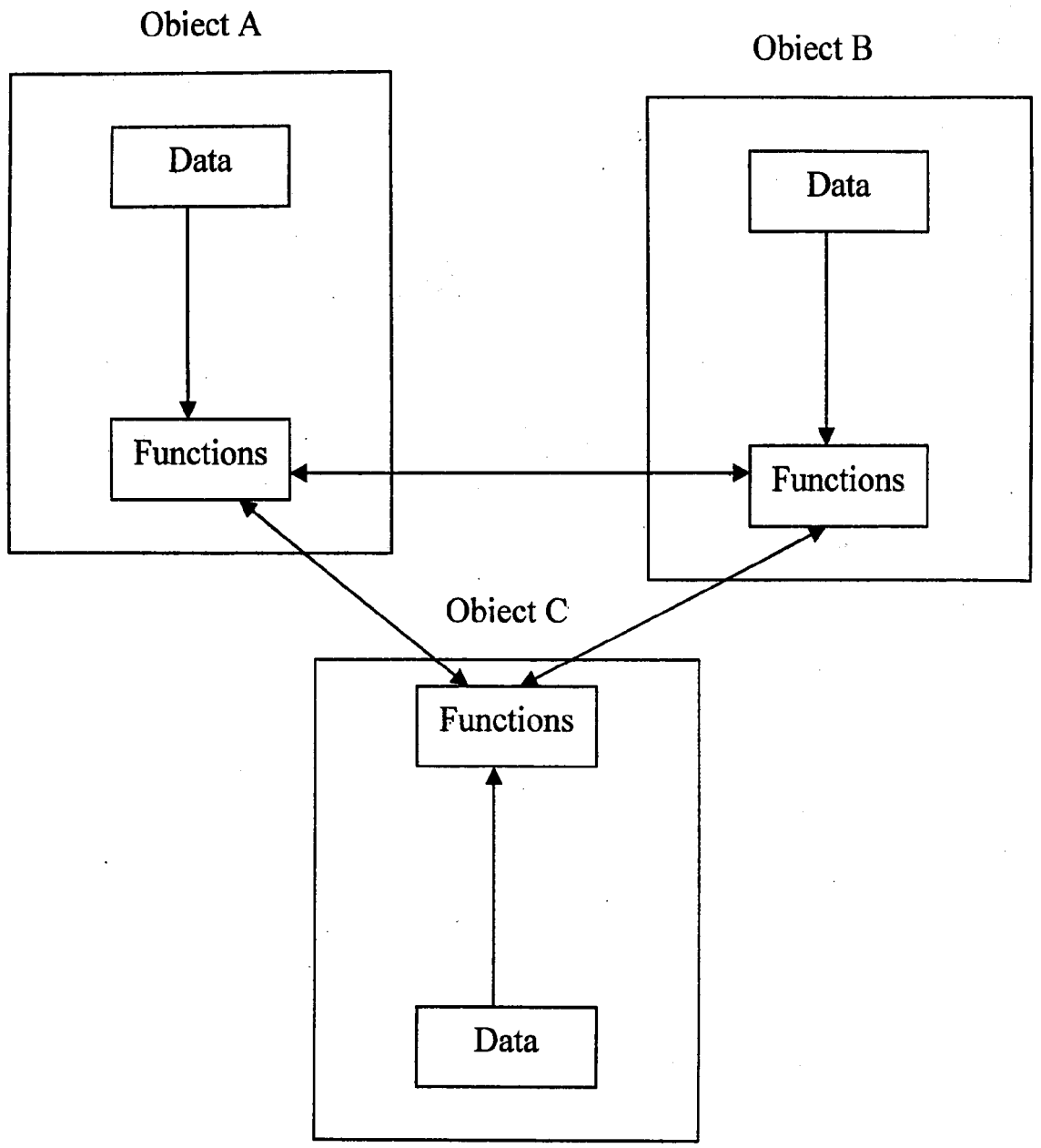
SPACE FOR HINT

OBJECT ORIENTED PROGRAMMING PARADIGM

The major motivating factor in the invention of object-oriented approach is to salvage some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allows it to flow freely around the system. It ties data more closely to the functions that operate on it and protects it from accidental modification from outside functions. OOP allows us to decompose a problem into a number of entries called objects and then builds data and functions around these entities. The organization of data and functions in object-oriented programs is follows.

The data of an object can be accessed only by the functions associated with that object. However, functions of one subject can access the functions of other subjects.

SPACE FOR HINT



Organization of data and functions in OOP

Some of the striking features of object-oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as *object.*
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people. It is therefore important to have a working definition of object-oriented programming before we proceed further. Our definition of

object-oriented programming is as follows : "Object-oriented programming is an approach that provides a way of modularizing programs by creating partitioned memory are for both data and functions that can be used as templates for creating copies of such modules on demand".

This is, an object is considered to be partitioned areas of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

SPACE FOR HINT

6.2 Basic Concepts of Object Oriented Programming

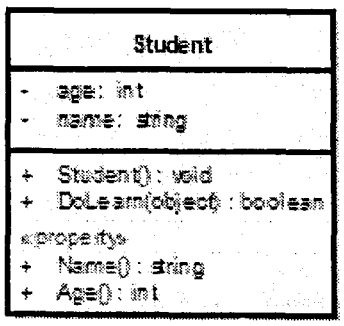
There are few principle concepts that form the foundation of object-oriented programming:

Object

An object can be considered a "*thing*" that can perform a set of **related** activities. The set of activities that the object performs defines the object's behavior. For example, the hand can grip something or a *Student* (*object*) can give the name or address.

In pure *OOP* terms an object is an instance of a class.

Class



A *class* is simply a representation of a type of *object*. It is the blueprint/ plan/ template that describe the details of an *object*. A class is the blueprint from which the individual objects are created. *Class* is composed of three things: a name, attributes, and operations.

```
public class Student
{
}
```

According to the sample given below we can say that the *student* object, named *objectStudent*, has created out of the *Student* class.

```
Student objectStudent = new Student();
```

SPACE FOR HINT

In real world, you'll often find many individual objects all of the same kind. As an example, there may be thousands of other bicycles in existence, all of the same make and model. Each bicycle has built from the same blueprint. In object-oriented terms, we say that the bicycle is an instance of the class of objects known as bicycles.

In the software world, though you may not have realized it, you have already used classes. For example, the *TextBox* control, you always used, is made out of the *TextBox* class, which defines its appearance and capabilities. Each time you drag a *TextBox* control, you are actually creating a new instance of the *TextBox* class.

Abstraction and Generalization

Abstraction is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail). The importance of abstraction is derived from its ability to hide irrelevant details and from the use of names to reference objects. Abstraction is essential in the construction of programs. It places the emphasis on what an object is or does rather than how it is represented or how it works. Thus, it is the primary means of managing complexity in large programs.

While abstraction reduces complexity by hiding irrelevant detail, generalization reduces complexity by replacing multiple entities which perform similar functions with a single construct. Generalization is the broadening of application to encompass a larger domain of objects of the same or different type. Programming languages provide generalization through variables, parameterization, generics and *polymorphism*. It places the emphasis on the similarities between objects. Thus, it helps to manage complexity by collecting individuals into groups and providing a representative which can be used to specify any individual of the group.

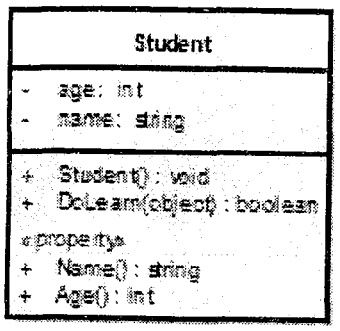
Abstraction and generalization are often used together. Abstracts are generalized through parameterization to provide greater utility. In parameterization, one or more parts of an entity are replaced with a name which is new to the entity. The name is used as a parameter. When the parameterized abstract is invoked, it is invoked with a binding of the parameter to an argument.

Encapsulation (or information hiding)

The encapsulation is the inclusion within a program object of all the resources need for the object to function - basically, the methods and the data. In *OOP* the encapsulation is mainly achieved by creating classes, the classes expose public methods and properties. The class is kind of a container or capsule or a cell, which encapsulate the set of methods, attribute and properties to provide its indented functionalities to other classes. In that sense, encapsulation also allows a class to change its internal implementation without hurting the overall functioning of the

system. That idea of encapsulation is to hide how a class does it but to allow requesting what to do.

SPACE FOR HINT



In order to modularize/ define the functionality of a one class, that class can uses functions/ properties exposed by another class in many different ways. According to Object Oriented Programming there are several techniques, classes can use to link with each other and they are named association, aggregation, and composition.

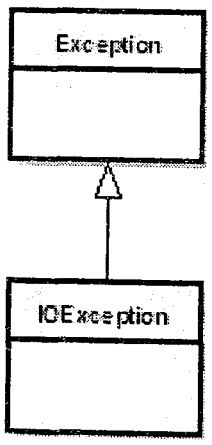
There are several other ways that an encapsulation can be used, as an example we can take the usage of an interface. The interface can be used to hide the information of an implemented class.

```
IStudent myStudent = new LocalStudent();
IStudent myStudent = new ForeignStudent();
```

According to the sample above (let's assume that *LocalStudent* and *ForeignStudent* are implemented by the *IStudent* interface) .

Inheritance

Ability of a new class to be created, from an existing class by extending it, is called *inheritance*.




```
public class Exception
{
}


public class IOException : Exception
```

SPACE FOR HINT

```
}
```

According to the above example the new class (*IOException*), which is called the derived class or subclass, inherits the members of an existing class (*Exception*), which is called the base class or super-class. The class *IOException* can extend the functionality of the class Exception by adding new types and methods and by overriding existing ones.

The inheritance is closely related with specialization. It is important to discuss those two concepts together with generalization to better understand and to reduce the complexity.

One of the most important relationships among objects in the real world is specialization, which can be described as the "**is-a**" relationship. When we say that a dog is a mammal, we mean that the dog is a specialized kind of mammal. It has all the characteristics of any mammal (it bears live young, nurses with milk, has hair), but it specializes these characteristics to the familiar characteristics of *canis domesticus*. A cat is also a mammal. As such, we expect it to share certain characteristics with the dog that are generalized in Mammal, but to differ in those characteristics that are specialized in cats.

The specialization and generalization relationships are both reciprocal and hierarchical. Specialization is just the other side of the generalization coin: Mammal generalizes what is common between dogs and cats, and dogs and cats specialize mammals to their own specific subtypes.

Similarly, as an example you can say that both *IOException* and *SecurityException* are of type Exception. They have all characteristics and behaviors of an Exception, That mean the *IOException* is a specialized kind of Exception. A *SecurityException* is also an *Exception*. As such, we expect it to share certain characteristic with *IOException* that are generalized in Exception, but to differ in those characteristics that are specialized in *SecurityExceptions*. In other words, Exception generalizes the shared characteristics of both *IOException* and *SecurityException*, while *IOException* and *SecurityException* specialize with their characteristics and behaviors.

In *OOP*, the specialization relationship is implemented using the principle called inheritance. This is the most common and most natural and widely accepted way of implement this relationship.

Polymorphisms

Polymorphisms is a generic term that means 'many shapes'. More precisely *Polymorphisms* means the ability to request that the same operations be performed by a wide range of different types of things.

In *OOP* the *polymorphisms* is achieved by using many different techniques named

SPACE FOR HINT

- method overloading,
- operator overloading and
- method overriding,

Polymorphism : Method Overloading

The method overloading is the ability to define several methods all with the same name.

```
public class MyLogger
{
   public void LogError(Exception e)
   {
      // Implementation goes here
   }

   public bool LogError(Exception e, string message)
   {
      // Implementation goes here
   }
}
```

Polymorphism : Operator Overloading

The operator overloading (less commonly known as ad-hoc *polymorphisms*) is a specific case of *polymorphisms* in which some or all of operators like +, - or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments.

```
public class Complex
{
   private int real;
   public int Real
   { get { return real; } }

   private int imaginary;
   public int Imaginary
   { get { return imaginary; } }

   public Complex(int real, int imaginary)
   {
      this.real = real;
      this.imaginary = imaginary;
   }

   public static Complex operator +(Complex c1, Complex c2)
```

SPACE FOR HINT

```
    {
      return new Complex(c1.Real + c2.Real, c1.Imaginary +
c2.Imaginary);
    }
}
```

In the above example we have overloaded the plus operator for adding two complex numbers. There the two properties named Real and Imaginary has been declared exposing only the required "*get*" method, while the object's constructor is demanding for mandatory real and imaginary values with the user defined constructor of the class.

Polymorphism : Method Overriding

Method overriding is a language feature that allows a subclass to override a specific implementation of a method that is already provided by one of its super-classes.

A subclass can give its own definition of methods but need to have the same signature as the method in its super-class. This means that when overriding a method the subclass's method has to have the same name and parameter list as the super-class's overridden method.

```
public class Complex
{
   private int real;
   public int Real
   { get { return real; } }

   private int imaginary;
   public int Imaginary
   { get { return imaginary; } }

   public Complex(int real, int imaginary)
   {
      this.real = real;
      this.imaginary = imaginary;
   }

   public static Complex operator +(Complex c1, Complex c2)
   {
     return new Complex(c1.Real + c2.Real, c1.Imaginary +
c2.Imaginary);
   }

   public override string ToString()
   {
      return (String.Format("{0} + {1}i", real, imaginary));
   }
}
```

In above example we have extended the implementation of the sample Complex class given under operator overloading section. This class has one overridden method named *"ToString"*, which override the default implementation of the standard *"ToString"* method to support the correct string conversion of a complex number.

SPACE FOR HINT

```
Complex num1 = new Complex(5, 7);
Complex num2 = new Complex(3, 8);

// Add two Complex numbers using the
// overloaded plus operator
Complex sum = num1 + num2;

// Print the numbers and the sum
// using the overriden ToString method
Console.WriteLine("({0}) + ({1}) = {2}", num1, num2, sum);
Console.ReadLine();
```

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Message Communication

An object-oriented program consists of set of objects that communicate with each other. The process of programming in an object-oriented language therefore involve the following basic steps:

1. Creating classes that define objects and their behavior
2. Creating objects from class definitions.
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes t easier to talk about building systems that directly model or simulate their real-world counterparts.

Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent example :

employee salary(name)

here employee is an object, salary is an message and name is an information.

SPACE FOR HINT

6.3 Benefits of OOP

OOP offers several benefits to both the program designed and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are :

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build program from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instance of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those objects in the program.
- It is easy to partition the work in a project based on objects.
- The data-centred design approach enables us to capture more details of a model in implementable form .
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external system much simpler.
- Software complexity can be easily managed.

Developing a software that is easy to use makes it hard to build. It is hoped that the object – oriented programming tools would help manage this problem.

6.4 Object-Oriented Languages

Object-oriented programming is not the right of any particular language. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially designed to supports the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object-based programming languages.
2. Object-oriented programming languages.

SPACE FOR HINT

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-based programming are :

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding.

Object-oriented programming incorporates all of object-based programming features along with two additional features, namely inheritance and dynamic binding.

OOP = Object – based features + inheritance + dynamic binding

Structure of a program

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:

```
// my first program in C++

#include <iostream>

int main ()
{
cout << "Hello World!";
return 0;
}
```

Output

```
Hello World!
```

The above line shows the source code for our first program. The output shows the result of the program once compiled and executed. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

SPACE FOR HINT

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream>

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive #include <iostream> tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it – the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them. Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

cout << "Hello World!";

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

SPACE FOR HINT

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

return 0;

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's preprocessor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into cout), which were all included within the block delimited by the braces ({}) of the main function.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of

```
int main ()
{
cout << " Hello World!";
return 0;
}
```

We could have written:

```
int main () { cout << "Hello World!"; return 0; }
```

All in just one line and this would have had exactly the same meaning as the previous code.

In C++, the separation between statements is specified with an ending semicolon (;) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of code in different lines serves only to make it more [illegible] schematic for the humans that may read it.

SPACE FOR HINT

Let us add an additional instruction to our first program:

```
// my second program in C++

#include <iostream>
int main ()
{
cout << "Hello World! ";
cout << "I'm a C++ program";
return 0;
}
```

Hello World! I'm a C++ program

In this case, we performed two insertions into cout in two different statements. Once again, the separation in different lines of code has been done just to give greater readability to the program, since main could have been perfectly valid defined this way:

```
int main () { cout << " Hello World! "; cout << " I'm a C++ program ";
return 0; }
```

We were also free to divide the code into more lines if we considered it more convenient:

```
int main ()
{
cout <<"Hello World!";
cout << "I'm a C++ program";
return 0;
}
```

And the result would again have been exactly the same as in the previous examples. Preprocessor directives (those that begin by #) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor and do not produce any code by themselves. Preprocessor directives must be specified in their own line and do not have to end with a semicolon (;).

Comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code. C++ supports two ways to insert comments:

// line comment

The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line. The second one, known as block comment, discards everything between the /*characters and the first appearance of the */ characters, with the possibility of including more than one line.

SPACE FOR HINT

We are going to add comments to our second program:

```
/* my second program in C++ with more comments */
#include <iostream>
int main ()
{
cout << "Hello World! "; // prints Hello World!
cout << "I'm a C++ program"; // prints I'm a C++ program
return 0;
}
```

Hello World! I'm a C++ program

If you include comments within the source code of your programs without using the comment characters combinations //, /* or */, the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.

Structure of a C++ Program

A typical C++ program would contain four sections. These section may be placed in separate code files and then compiled independently or ointly.

Include files

Class declaration

Class functions

Definitions

Main function programs

It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member function definition). Finally, the main program that uses the calls is placed in a third file which "includes" the previous two files as well as any other files required.

SPACE FOR HINT

This approach is based on the concept of client-server model. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

Example

```
#include <iostream.h>

class person                              // NEW DATA TYPE
{
        char name[30];
        int age;
        public:
                void getdata(void);
                void display(void);
};

void person :: getdata(void)                    // MEMBER FUNCTION
{
        cout<<"Enter name : ";
        cin>> name;
        cout<<"Enter age :";
        cin>> age;
}

void person :: display(void)       // MEMBER FUNCTION
{
        cout>> "\nName : " << name;
        cout>> "\Age : " << age;

}

main()
{
        person p;                  // OBJECT OF TYPE person
        p.getdata();
        p.display();
}
```

Variables. Data Types.

The usefulness of the "Hello World" programs shown in the previous section is quite questionable. We had to write several lines of code, compile them, and then execute the resulting program just to obtain a simple sentence written on the screen as result. It certainly would have been much faster to type the output sentence by ourselves.

However, programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work we need to introduce the concept of variable.

SPACE FOR HINT

Let us think that I ask you to retain the number 5 in your mental memory, and then I ask you to memorize also the number 2 at the same time. You have just stored two different values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now for example subtract and obtain 4 as result.

The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

```
a = 5;
b = 2;
a = a + 1;
result = a - b;
```

Obviously, this is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them.

Therefore, we can define a variable as a portion of memory to store a determined value. Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were a, b and result, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

REFERENCE VARIABLES

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable. For example, if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent that variable. A reference variable is created as follows :

data-type & reference-name = variable-name

Example :

```
float total = 100;
float & sum=total;
```

total is a float type variable that has already been declared. Sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object in the memory. Now, the statements

SPACE FOR HINT

```
cout<<total;
```

and

```
cout<<sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise the assignment

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declarations. This establishes the correspondence between the reference and the data object that it names. Note that the initialization of a reference variable is completely different from assignment to it.

6.5 TOKENS

6.5.1 Identifiers

A valid identifier is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (_), but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case they can begin with a digit.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are *reserved keywords*.

6.5.2. Keywords

The standard reserved keywords are:

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete,
do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto,if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register,reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

SPACE FOR HINT

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Very important: The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the **RESULT** variable is not the same as the **result** variable or the Result variable. These are three different variable identifiers.

Fundamental data types

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Table : Variable Types and the storage size

Type	Size	Values
unsigned short int	2 bytes	0 to 65,535
short int	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int	4 bytes	-2,147,483,648 to 2,147,483,647
int (16 bit)	2 bytes	-32,768 to 32,767
int (32 bit)	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int (16 bit)	2 bytes	0 to 65,535
unsigned int (32 bit)	2 bytes	0 to 4,294,967,295
char	1 byte	256 character values
float	4 bytes	1.2e-38 to 3.4e38
double	8 bytes	2.2e-308 to 1.8e308

The sizes of variables might be different from those shown in the above depending on the compiler and the computer you are using.

SPACE FOR HINT

6.5.3 Declaration of variables

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier.

For example:

```
int a;
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once declared, the variables a and mynumber can be used within the rest of their scope in the program. If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

This declares three variables (a, b and c), all of them of type int, and has exactly the same meaning as:

```
int a;
int b;
int c;
```

The integer data types char, short, long and int can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier signed or the specifier unsigned before the type name. For example:

```
unsigned short int NumberOfSisters;
```

```
signed int MyAccountBalance;
```

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

int MyAccountBalance; with exactly the same meaning (with or without the keyword signed) An exception to this general rule is the char type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters. You should use either signed or unsigned if you intend to store numerical values in a char-sized variable. short and long can be used alone as type

specifiers. In this case, they refer to their respective integer fundamental types: short is equivalent to short int and long is equivalent to long int. The following two variable declarations are equivalent:

SPACE FOR HINT

```
short Year;
short int Year;
```

Finally, signed and unsigned may also be used as standalone type specifiers, meaning the same as signed int and unsigned int respectively. The following two declarations are equivalent:

```
unsigned NextYear;
unsigned int NextYear;
```

To see what variable declarations look like in action within a program.

```
// operating with variables
#include <iostream>
int main ()
{
// declaring variables:
int a, b;
int result;
// process:
a = 5;
b = 2;
a = a + 1;
result = a - b;
// print out the result:
cout << result;
// terminate the program:
return 0;
}
4
```

Scope of variables

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that a, b, and result were of type int.

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block. Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

The scope of local variables is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the

SPACE FOR HINT

beginning of the body of a function (like in function main) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to main, the local variables declared in main could not be accessed from the other function and vice versa.

Initialization of variables

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

type identifier = initial_value ;

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

int a = 0;

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (()):

type identifier (initial_value) ;

For example:

int a (0);

Both ways of initializing variables are valid and equivalent in C++.

```
// initialization of variables
#include <iostream>
int main ()
{
int a=5; // initial value = 5
int b(2); // initial value = 2
int result; // initial value
undetermined
a = a + 3;
result = a - b;
cout << result;
return 0;
}
6
```

Introduction to strings

SPACE FOR HINT

Variables that can store non-numerical values that are longer than one single character are known as strings. The C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: <string> and have access to the std namespace (which we already had in all our previous programs thanks to the using namespace statement).

```
// my first string
#include <iostream>
#include <string>
int main ()
{
string mystring = "This is a string";
cout << mystring;
return 0;
}
```

Output
This is a string

As you may see in the previous example, strings can be initialized with any valid string literal just like numerical type variables can be initialized to any valid numerical literal. Both initialization formats are valid with strings:

```
string mystring = "This is a string";
string mystring ("This is a string");
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution:

```
// my first string
#include <iostream>
#include <string>
int main ()
{
string mystring;
mystring = "This is the initial string content";
cout << mystring << endl;
mystring = "This is a different string content";
cout << mystring << endl;
return 0;
}
```

SPACE FOR HINT

output

This is the initial string content
This is a different string content

Constants

Constants are expressions with a fixed value.

Literals

Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

the 5 in this piece of code was a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Integer Numerals

```
1776
707
-273
```

They are numerical constants that identify integer decimal values. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write 1776 in a program, we will be referring to the value 1776. In addition to decimal numbers (those that all of us are used to use every day) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (zero character). And in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75 // decimal
0113 // octal
0x4b // hexadecimal
```

All of these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively. Literal constants, like variables, are considered to have a specific data type. By default, integer literals are of type int. However, we can force

them to either be unsigned by appending the u character to it, or long by appending l:

75 // int
75u // unsigned int
75l // long
75ul // unsigned long

In both cases, the suffix can be specified using either upper or lowercase letters.

Floating Point Numbers

They express numbers with decimals and/or exponents. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character:

3.14159 // 3.14159
6.02e23 // 6.02 x 10^23
1.6e-19 // 1.6 x 10^-19
3.0 // 3.0

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number three expressed as a floating-point numeric literal. The default type for floating point literals is double. If you explicitly want to express a float or long double numerical literal, you can use the f or l suffixes respectively:

3.14159L // long double
6.02e23f // float

Any of the letters that can be part of a floating-point numerical constant (e, f, l) can be written using either lower or uppercase letters without any difference in their meanings.

Character and string literals

There also exist non-numerical constants, like:
'z'
'p'
"Hello world"
"How do you do?"

The first two expressions represent single character constants, and the following two represent string literals composed of several characters. Notice that to represent a single character we enclose it between single

SPACE FOR HINT

SPACE FOR HINT

quotes (') and to express a string (which generally consists of more than one character) we enclose it between double quotes (").

When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

```
x
'x'
```

x alone would refer to a variable whose identifier is x, whereas 'x' (enclosed within single quotation marks) would refer to the character constant 'x'.

Character and string literals have certain peculiarities, like the escape codes. These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t). All of them are preceded by a backslash (\). Here you have a list of some of such escape codes:

\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Additionally, you can express any character by its numerical ASCII code by writing a backslash character (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A). String literals can extend to more than a single line of code by putting a backslash sign (\) at the end of each unfinished line.

```
"string expressed in \ two lines"
```

You can also concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character:

"this forms" "a single" "string" "of characters"

Finally, if we want the string literal to be explicitly made of wide characters (wchar_t), instead of narrow characters(char), we can precede the constant with the L prefix: L"This is a wide character string" Wide characters are used mainly to represent non-English or exotic character sets.

Boolean literals

There are only two valid Boolean values: true and false. These can be expressed in C++ as values of type bool by using the Boolean literals true and false.

Defined constants (#define)

You can define your own names for constants that you use very often without having to resort to memory consuming variables, simply by using the #define preprocessor directive. Its format is:

#define identifier value

For example:
#define PI 3.14159
#define NEWLINE '\n'

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

```
// defined constants: calculate circumference
#include <iostream>
#define PI 3.14159
#define NEWLINE '\n'
int main ()
{
double r=5.0; // radius
double circle;
circle = 2 * PI * r;
cout << circle;
cout << NEWLINE;
return 0;
}
```

31.4159

SPACE FOR HINT

SPACE FOR HINT

In fact the only thing that the compiler preprocessor does when it encounters #define directives is to literally replace any occurrence of their identifier (in the previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159 and '\n' respectively). The #define directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end. If you append a semicolon character (;) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

Declared constants (const)

With the const prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;
const char tabulator = '\t';
```

Here, pathwidth and tabulator are two typed constants. They are treated just like regular variables except that their values cannot be modified after their definition.

6.5.4 OPERATORS

C++ has a rich set of operators. All C operators are valid in c++ also. In addition, C++ introduces some new operators.

::	Scope resolution operator
::*	Pointer-to-member declarator
->*	Pointer-to-member operator
.*	Pointer-to-member operator
delete	Memory release operator
endl	Line feed operator
new	Memory allocation operator
setw	Field width operator

Scope Resolution operator

Like C, C++ is also a block-structured language. Blocks and scope can be used to constructing programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block.

SPACE FOR HINT

```
{
        int x = 10;
        ……..
        …….
}
……..
…….
{
        int x = 1;
        ……..
        …….
}
```

The two declaration of x refer to two different memory location containing different values. Statement in the second block cannot refer to the variable x declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style is common.

```
……..
…….
{
        int x = 10;
        ……..
        …….
{
                int x = 1;
                ……..
                …….
        }
}       ……
```

Block2 is contained in block1. Note that a declaration in an inner block hider a declaration of the same variable in an outer block and therefore, each declaration of x causes it to refer to a different data object. Within the inner block, the variable x will refer to the data object declared therein.

In C, the global version of a variable cannot be accessed from within inner block. C++ resolves this problem by introducing a new operator :: is called the ***scope resolution operator.*** This can be used to uncover a hidden variable. It takes the following form

:: variable-name

This operator allows access to the global version of a variable. For example, ::count means the global version of the variable count (and not the local variable count declared in that block)

Example program

SPACE FOR HINT

```
#include<iostream.h>
main()
{
        int m = 20;
        {
                int  k = m;
                int m = 30;
                cout<< "we are in inner block\n";
                cout<< "k=   " << k << "\n";
                cout<< "m=   " << m << "\n";
                cout<< "::m=   " << ::m <<"\n";
        }
        cout  << "\n We are in outer block \n";
        cout<< "m=   " << m << "\n";
        cout<< "::m=   " << ::m <<"\n";
```

output

```
        We are in inner block
        k=20
        m=30
        ::m = 10

        We are in outer block
        m=20
        ::m = 10
```

In the above program, the variable m is declared at three places, namely, outside the main() function, inside the main() and inside the inner block. Note that ::m will always refer to the global m. In the inner block ::m refers to the value 10 and not 20.

Memory Management operators

C uses malloc() and calloc() function to allocate memory dynamically at run time. Similarly, it uses the function free ( ) to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ supports these functions, it also defines two unary operators new and delete that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as free store operators.

Remember that an object can be created by using new and destroyed by using delete as and when required. That is, the lifetime of an object is directly under our control and is unrelated to the block structure of the program. A data object created inside a block with new will remain in existence until it is explicitly destroyed by using delete.

SPACE FOR HINT

The new operator can be used to create objects of any type. It takes the following general form:

pointer-variable = new data-type;

Here, pointer-variable is a pointer of type data-type. The new operator allocates sufficient memory to hold a data object of type data-type and returns the address of the object. The data-type may be any valid data type. The pointer-variable holds the address of the memory space allocated.
Examples:

```
p= new int;
q= new float;
```

where p is a pointer of type int and q is a pointer of type float. Remember, p and q must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int * p = new int;
float * q = new float;
```

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created int object and7.5 to the float object.

We can also initialize the memory using the new operator. This is done as follows:

Pointer-variable = new data-type(value);

Here, value specifies the initial value. Examples:

```
int * p = new int (25);
float * q = new float (7.5);
```

As mentioned earlier, new can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is:

Pointer-variable = new data-type [size];

Here, size specifies the number of elements in the array. For example, the statement

```
Int * p = new int [10];
```

Creates a memory space for an array of 10 integers. p[0] will refer to the first element, p[1] to the second element, and so on.

SPACE FOR HINT

When creating multi-dimensional arrays with new, all the array sizes must be supplied.

```
array_ptr = new int[3][5][4];
array_ptr = new int [m][5][4];
```

The first dimension may be a variable whose value is supplied at runtime. All others must be constants.
When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
delete pointer-variable;
```

The pointer-variable is the pointer that points to a data object created woth new. Examples:

```
delete p;
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of delete:

```
delete [size] pointer-variable;
```

The size specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ does not require the size to be specified. For example,

```
delete [ ] p;
```

will delete the entire array pointed to by p.

If sufficient memory is not available for allocation happens, in such cases, like malloc( ), new returns a null pointer. Therefore, it may be a good idea to check for the pointer produced by new before using it. It is done as follows:

```
....
....
p = new int;
if(!p)
{
   cout <<"allocation failed \n";
}
....
....
```

The new operator has several advantages over the function malloc( ).

1. It automatically computes the size of the data object. We need

SPACE FOR HINT

2. It automatically returns the correct pointer type, so that is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, new and delete can be overloaded.

MANIPULATORS

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw.**

The endl manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n". For example, the statement

```
.....
.....
cout <<"m= " <<m<<endl
cout <<"n= " <<n<<endl
cout <<"p= " <<p<<endl
.....
.....
```

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597, 14 and 175 respectively, the output will appear as shown below :

```
m = 2597
n = 14
p = 175
```

It is not the ideal output, it should be appear rather as follows:

```
m = 2597
n =   14
p =  175
```

The **setw** manipulators does this job. It is used as follows :

```
cout<<setw(5)<<sum<<endl;
```

The manipulator setw(5) specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below

		3	4	5

Example program :

```
#include<iostream.h>
#include<iomanip.h>
main()
```

SPACE FOR HINT

```
{
        int Basic = 950, Allowance = 95, Total = 1045;

        cout <<setw(10)<<"Basic"<<setw(10)<<Basic<<endl
        cout<<setw(10)<<"Allowance"<<setw(10)<<Allowance<<endl
        cout<<setw(10)<<"Total"<<setw(10)<<Total<<endl;

}
```

Output

```
Basic          950
Allowance       95
Total         1045
```

Note that the character strings are also printed right-justified.

TYPE CAST OPERATOR

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

Assignment operator (=)

The assignment operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a

variable whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these.

SPACE FOR HINT

The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:

```
a = b;
```

This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost. Consider also that we are only assigning the value of b to a at the moment of the assignment operation. Therefore a later change of b will not affect the new value of a.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator
#include <iostream>
using namespace std;
int main ()
{
int a, b; // a:?, b:?
a = 10; // a:10, b:?
b = 4; // a:10, b:4
a = b; // a:4, b:4
b = 7; // a:4, b:7
cout << "a:";
cout << a;
cout << " b:";
cout << b;
return 0;
}
a:4 b:7
```

This code will give us as result that the value contained in a is 4 and the one contained in b is 7. Notice how a was not affected by the final modification of b, even though we declared a = b earlier (that is because of the *right-toleft rule*).

A property that C++ has over other programming languages is that the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation. For example:

```
a = 2 + (b = 5);
```
is equivalent to:
```
b = 5;
a = 2 + b;
```

SPACE FOR HINT

that means: first assign 5 to variable b and then assign to a the value 2 plus the result of the previous assignment of b (i.e. 5), leaving a with a final value of 7.

The following expression is also valid in C++:
a = b = c = 5;

It assigns 5 to the all the three variables: a, b and c.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:

+ addition
- subtraction
* multiplication
/ division
% modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

a = 11 % 3;

the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=,^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression is equivalent to

value += increase; value = value + increase;
a -= 5; a = a - 5;
a /= b; a = a / b;
price *= units + 1; price = price * (units + 1);
and the same for all other operators. For example:

```
// compound assignment operators
#include <iostream>
int main ()
{
int a, b=3;
```

```
a = b;
a+=2; // equivalent to a=a+2
cout << a;
return 0;
}
5
```

SPACE FOR HINT

Increment and Decrement Operator++, --)

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
c++;
c+=1;
c=c+1;
```

are all equivalent in its functionality: the three of them increase by one the value of c.
In the early C compilers, the three previous expressions probably produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally done automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

```
B=3;
A=++B;
// A contains 4, B contains 4
```

```
B=3;
A=B++;
// A contains 3, B contains 4
```

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

SPACE FOR HINT

Example 1

```
#include <iostream.h>
int main()
{
 int    val = 1;
        cout << "\nval is " << val++ << " and then post-incremented\n";
        cout << "val is now " << val << "\n";
        cout << "val is pre-incremented to " << ++val << '\n';
        return 0;
```

Relational and equality operators (==, !=, >, <, >=, <=)

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

== Equal to
!= Not equal to
> Greater than
< Less than
>= Greater than or equal to
<= Less than or equal to
Here there are some examples:
(7 == 5) // evaluates to false.
(5 > 4) // evaluates to true.
(3 != 2) // evaluates to true.
(6 >= 6) // evaluates to true.
(5 < 5) // evaluates to false.

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that a=2, b=3 and c=6,

(a == 5) // evaluates to false since a is not equal to 5.
(a*b >= c) // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c) // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a) // evaluates to true.

Be careful! The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression ((b=2) == a), we first assigned the value 2 to b and then we compared it to a, that also stores the value 2, so the result of the operation is true.

SPACE FOR HINT

Logical operators (!, &&, ||)

The Operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

!(5 == 5) // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4) // evaluates to true because (6 <= 4) would be false.
!true // evaluates to false
!false // evaluates to true.

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

&& OPERATOR

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:

|| OPERATOR

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

For example:

((5 == 5) && (3 > 6)) // evaluates to false (true && false).
((5 == 5) || (3 > 6)) // evaluates to true (true || false).

Conditional operator (?)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as

SPACE FOR HINT

false. Its format is: condition ? result1 : result2 If condition is true the expression will return result1, if it is not it will return result2.

```
7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
5>3 ? a : b // returns the value of a, since 5 is greater than 3.
a>b ? a : b // returns whichever is greater, a or b.
```

```
// conditional operator
#include <iostream>
int main ()
{
int a,b,c;
a=2;
b=7;
c = (a>b) ? a : b;
cout << c;
return 0;
}
```

```
7
```

In this example a was 2 and b was 7, so the expression being evaluated (a>b) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was b, with a value of 7.

Comma operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```

Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator and equivalent description

OR	Bitwise Inclusive OR
^ X OR	Bitwise Exclusive OR
~ NOT	Unary complement (bit inversion)
<< SHL	Shift Left
>> SHR	Shift Right

sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to a because char is a one-byte long type. The value returned by sizeof is a constant, so it is always determined before program execution.

7. Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compoundstatement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }:

```
{ statement1; statement2; statement3; }
```

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({}). But in the case that we want the statement to be a compound statement it must be enclosed between braces ({}), forming a block.

Conditional structure: if and else

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

```
if (condition) statement
```

Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored

SPACE FOR HINT

(not executed) and the program continues right after this conditional structure.

For example, the following code fragment prints x is 100 only if the value stored in the x variable is indeed 100:

```
if (x == 100)
cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

```
if (x == 100)
{
cout << "x is ";
cout << x;
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

```
if (condition) statement1 else statement2
```

For example:

```
if (x == 100)
cout << "x is 100";
else
cout << "x is not 100";
```

prints on the screen x is 100 if indeed x has a value of 100, but if it has not -and only if not- it prints out x is not 100. The if + else structures can be concatenated with the intention of verifying a range of values.

The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
cout << "x is positive";
else if (x < 0)
cout << "x is negative";
else
cout << "x is 0";
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

Example 1

SPACE FOR HINT

```
#include <iostream.h>
int main()
{
  int ascii_val;

        for (ascii_val = 32; ascii_val < 256; ascii_val++)
        {
                cout << '\t' << (char)ascii_val;
                if (ascii_val % 9 == 0)
                        cout << '\n';
  }

  return 0;
}
```

Example 2

```
#include <iostream.h>
int main()
{
        int  your_number;
        cout << "Enter a whole number: ";
        cin >> your_number;

        if (your_number % 2 == 0)
          cout << "Your number is even\n";
        else
      {
          cout << "Your number is odd.\n";
        }
cout << "That's all!\n";
return 0;
}
```

Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

while (expression) statement

and its functionality is simply to repeat statement while the condition set in expression is true. For example, we are going to make a program to countdown using a while-loop:

SPACE FOR HINT

```
// custom countdown using while
#include <iostream>
int main ()
{
int n;
cout << "Enter the starting number > ";
cin >> n;
while (n>0)
{
cout << n << ", ";
--n;
}
cout << "FIRE!\n";
return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition n>0 (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition (n>0) remains being true.

The do-while loop

Its format is:

```
do statement while (condition);
```

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

```
// number echoer
#include <iostream>
int main ()
{
unsigned long n;
do {
cout << "Enter number (0 to end): ";
cin >> n;
cout << "You entered: " << n << "\n";
} while (n != 0);
return 0;
}
```

```
Enter number (0 to end): 12345
You entered: 12345
```

SPACE FOR HINT

```
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined withinthe loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

Example 1

```
#include <conio.h>
#include <ctype.h>
#include <iostream.h>
int main()
{
  char cmd;

  do
{
    cout << "Chart desired: Pie Bar Scatter Line Three-D Exit";
    cout << "\nPress first letter of the chart you want: ";
    cmd = toupper(getch());
    cout << '\n';

    switch (cmd)
    {
      case 'P': cout << "Doing pie chart\n"; break;
      case 'B': cout << "Doing bar chart\n"; break;
      case 'S': cout << "Doing scatter chart\n"; break;
      case 'L': cout << "Doing line chart\n"; break;
      case 'T': cout << "Doing 3-D chart\n"; break;
      case 'E': break;
      default : cout << "Invalid choice. Try again\n";
    }
  } while (cmd != 'E');

  return 0;
}
```

SPACE FOR HINT

The for loop

Its format is:

for (initialization; condition; increase) statement;

and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration. It works in the following way:

1. Initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.

2. Condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).

3. Statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.

4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream>
int main ()
{
for (int n=10; n>0; n--)
{
cout << n << ", ";
}
cout << "FIRE!\n";
return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before). Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in in[illegible]ization, for example. The comma [illegible] to separate

For example, suppose that we wanted to initialize more than one variable in our loop:

SPACE FOR HINT

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
// whatever here...
}
```

This loop will execute for 50 times if neither n or i are modified within the loop: n starts with a value of 0, and i with 100, the condition is n!=i (that n is not equal to i). Because n is increased by one and i decreased by one, the loop's condition will become false after the 50th loop, when both n and i will be equal to 50.

Example 1

```
#include <iostream.h>
int main()
{
        int number, total;
        for (number = 1, total = 0; number < 11; total += number,
        number++);
        cout << "\nTotal of numbers from 1 to 10 is "<< total;

  return 0;
}
```

Jump statements.

The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

```
// break loop example
#include <iostream>
int main ()
{
int n;
for (n=10; n>0; n--)
{
cout << n << ", ";
if (n==3)
{
cout << "countdown aborted!";
break;
}
```

SPACE FOR HINT

```
return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// continue loop example
#include <iostream>
int main ()
{
for (int n=10; n>0; n--)
{
if (n==5) continue;
cout << n << ", ";
}
cout << "FIRE!\n";
return 0;
}
```

10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

Example 1

```
#include <iostream.h>
int main()
{
  int num = 0;
  while (num++ <= 10)
  {
    if (num % 2 != 0)
      continue;
    cout << '\n' << num;
  }

  return 0;
}
```

The goto statement

goto allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations. The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

SPACE FOR HINT

```
// goto loop example
#include <iostream>
int main ()
{
int n=10;
loop:
cout << n << ", ";
n--;
if (n>0) goto loop;
cout << "FIRE!\n";
return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

The exit function

exit is a function defined in the cstdlib library.

The purpose of exit is to terminate the current program with a specific exit code. Its prototype is:

```
void exit (int exitcode);
```

The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

The selective structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several if and else if instructions. Its form is the following:

```
switch (expression)
{
case constant1:
group of statements 1;
break;
case constant2:
group of statements 2;
break;
```

SPACE FOR HINT

```
.
.
default:
default group of statements
}
```

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure. If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

switch example if-else equivalent

```
switch (x) {
case 1:
cout << "x is 1";
break;
case 2:
cout << "x is 2";
break;
default:
cout << "value of x unknown";
}
if (x == 1) {
cout << "x is 1";
}
else if (x == 2) {
cout << "x is 2";
}
else {
cout << "value of x unknown";
}
```

The switch statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached. For example, if we did not include a break statement after the first group for case one,

the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated.

SPACE FOR HINT

Example 1

```
#include <conio.h>
#include <ctype.h>
#include <iostream.h>
int main()
{
  char cmd;

  do {
    cout << "Chart desired: Pie Bar Scatter Line Three-D Exit";
    cout << "\nPress first letter of the chart you want: ";
    cmd = toupper(getch());
    cout << '\n';

    switch (cmd)
    {
      case 'P': cout << "Doing pie chart\n"; break;
      case 'B': cout << "Doing bar chart\n"; break;
      case 'S': cout << "Doing scatter chart\n"; break;
      case 'L': cout << "Doing line chart\n"; break;
      case 'T': cout << "Doing 3-D chart\n"; break;
      case 'E': break;
      default : cout << "Invalid choice. Try again\n";
    }
  } while (cmd != 'E');

  return 0;
}
```

C.Y.P

1. What is OOP ?

2. Explain the Basic Concepts of OOP?

3. What are the benefits of OOP?

4. What refers tokens in C++ ?

5. What is the purpose of manipulators ? Explain any two manipulators

Table of contents

UNIT – 7

SPACE FOR HINT

7. INTRODUCTION - FUNCTIONS

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is, in effect, a subprogram that can act on data and return a value. Every C++ program has at least one function, main(). When your program starts, main() is called automatically. main() might call other functions, some of which might call still others.

Each function has its own name, and when that name is encountered, the execution of the program branches to the body of that function. When the function returns, execution resumes on the next line of the calling function.

When a program calls a function, execution switches to the function and then resumes at the line after the function call. Well-designed functions perform a specific and easily understood task. Complicated tasks should be broken down into multiple functions, and then each can be called in turn.

Functions come in two varieties: user-defined and built-in. Built-in functions are part of your compiler package--they are supplied by the manufacturer for your use.

7.1 THE MAIN FUNCTION

ANSI C does not specify any return type for the main() function which is the starting point for the execution of a program. The definition of main() would look like this:

```
main()
{
        //main program statements
}
```

This is perfectly valid because the main() in ANSI.C does not return any value.

In C++, the main() returns a value of type int to the operating system. C++ therefore, explicitly define main() as matching one of the following prototypes:

```
int main();
int main(int argc, char* argv[]);
```

SPACE FOR HINT

The functions that have a return value should use the return statement for termination. The main() function in C++ is, therefore, defined as follows:

```
int main()
{
        ....
        ....
        return(0);
}
```

Since the return type of functions is int by default, the keyword int in the main() header is optional. Most C++ compilers will generate any error or warning if there is no return statement. Turbo C++ issues the warning.

```
Function should return a value
```

and then proceeds to compile the program. It is good programming practice to actually return a value from main().

Many operating system test the return value (called exit value) to determine if there is any problem. The normal convention is that an exit value of zero means the program ran successfully, while a nonzero value means there was problem. The explicit use of a return(0) statement will indicate that the program was successfully executed.

7.1.1. Declaring and Defining Functions

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

type name (parameter1, parameter2, ...) { statements }

where:
• type is the data type specifier of the data returned by the function.
• name is the identifier by which it will be possible to call the function.
• parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
• statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:

Example 1

```
// function example
#include <iostream>
```

```
int addition (int a, int b)
{
int r;
=a+b;
eturn (r);

t main ()

t z;
= addition (5,3);
ut << "The result is " << z;
turn 0;
```

ıe result is 8

order to examine this code, first of all remember something said at the ginning of this tutorial: a C++ program always begins its execution by : main function. So we will begin there. We can see how the main ıction begins by declaring the variable z of type int. Right after that, : see a call to a function called addition. Paying attention we will be le to see the similarity between the structure of the call to the function d the declaration of the function itself some code lines above:

ıe parameters and arguments have a clear correspondence. Within the ıin function we called to addition passing two values: 5 and 3, that rrespond to the int a and int b parameters declared for function dition.

the point at which the function is called from within main, the control lost by main and passed to function addition. The value of both guments passed in the call (5 and 3) are copied to the local variables t a and int b within the function.

ınction addition declares another local variable (int r), and by means of expression r=a+b, it assigns to r the result of a plus b. Because the ual parameters passed for a and b are 5 and 3 respectively, the result 3.

ıe following line of code:

turn (r);

nalizes function addition, and returns the control back to the function ıat called it in the first place (in this case, main). At this moment the ogram follows it regular course from the same point at which it was ıterrupted by the call to addition. But additionally, because the return atement in function addition specified a value: the content of variable r eturn (r);), which at that moment had a value of 8. This value becomes ıe value of

SPACE FOR HINT

SPACE FOR HINT

evaluating the function call.

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable z will be set to the value returned by addition (5, 3), that is 8. To explain it another way, you can imagine that the call to a function (addition (5,3)) is literally replaced by the value it returns (8).

The following line of code in main is:

cout << "The result is " << z;

That, as you may already expect, produces the printing of the result on the screen.

Example 2

```
#include <iostream.h>
float tax (float) ;
int main()
{
        float purchase, tax_amt, total;
        cout << "\nAmount of purchase: ";
        cin >> purchase;

        tax_amt = tax(purchase);
        total = purchase + tax_amt;
        cout.precision(2);
        cout << "\nPurchase is: " << purchase;
        cout << "\nTax: " << tax_amt;
        cout << "\nTotal: " << total;

        return 0;
}

float tax (float amount)
   {
      float rate = 0.065;
        return(amount * rate);
   }
```

7.1.2 Functions with no type. The use of void.

If you remember the syntax of a function declaration:

type name (argument1, argument2 ...) statement

you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the

function with the return statement). But what if we want to return no value?

SPACE FOR HINT

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the function. This is a special specifier that indicates absence of type.

```
// void function example
#include <iostream>
void printmessage ()
{
cout << "I'm a function!";
}
int main ()
{
printmessage ();
return 0;
}
I'm a function!
```

void can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function printmessage could have been declared as:

```
void printmessage (void)
{
cout << "I'm a function!";
}
```

Although it is optional to specify void in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to printmessage is:

```
printmessage ();
```

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect: printmessage;

7.2 Function Prototyping

Function prototyping is one of the major improvements added to C++

SPACE FOR HINT

by giving details such as the number and type of arguments and the typ of return values. With function prototyping, a template is always use when declaring and defining a function. When a function is called, th compiler uses the template to ensure that proper arguments are passec and the return value is treated correctly. Any violation in matching th arguments or the return types will be caught by the compiler at the tim of compilation itself. These checks and controls did not exist in th conventional C functions.

Function prototype is a declaration statement in the calling program an is of the following form:

type function-name (argument-list);

The argument-list contains the types and names of arguments that mus be passed to the function. Example:

float volume(int x, float y, float z);

Until now, we have defined all of the functions before the firs appearance of calls to them in the source code. These calls wer generally in function main which we have always left at the end of th source code. If you try to repeat some of the examples of function described so far, but placing the function main before any of the othe functions that were called from within it, you will most likely obtair compiling errors. The reason is that to be able to call a function it mus have been declared in some earlier point of the code, like we have done in all our examples.

But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces { }) and instead of that we end the prototype declaration with a mandatory semicolon (;).

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called protofunction with two int parameters with any of the following declarations:

int protofunction (int first, int second);

```
int protofunction (int, int);
```

SPACE FOR HINT

Anyway, including a name for each variable makes the prototype more legible.

```
// declaring functions prototypes
#include <iostream>
void odd (int a);
void even (int a);
int main ()
{
int i;
do {
cout << "Type a number (0 to exit): ";
cin >> i;
odd (i);
} while (i!=0);
return 0;
}
void odd (int a)
{
if ((a%2)!=0) cout << "Number is odd.\n";
else even (a);
}
void even (int a)
{
if ((a%2)==0) cout << "Number is even.\n";
else odd (a);
}
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.
```

This example is indeed not an example of efficiency. We are sure that at this point we can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors. The first things that we see are the declaration of functions odd and even:

```
void odd (int a);

void even (int a);
```

SPACE FOR HINT

This allows these functions to be used before they are defined, for example, in main, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because in odd there is a call to even and in even there is a call to odd. If none of the two functions had been previously declared, a compilation error would happen, since either odd would not not be visible from even (because it has still not been declared), or even would not be visible from odd (for the same reason).

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

7.3 FUNCTIONS CALL BY REFERENCE

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```
int x=5, y=3, z;

z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves. This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called. But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

```
// passing parameters by reference
#include <iostream>

void duplicate (int& a, int& b, int& c)
{
a*=2;
```

```
}
int main ()
{
int x=1, y=3, z=7;
duplicate (x, y, z);
cout << "x=" << x << ", y=" << y << ", z=" << z;
return 0;
}
```

x=2, y=6, z=14

SPACE FOR HINT

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z. That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled. If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
#include <iostream>
void prevnext (int x, int& prev, int& next)
{
prev = x-1;
next = x+1;
```

SPACE FOR HINT

```
}
int main ()
{
int x=100, y, z;
prevnext (x, y, z);
cout << "Previous=" << y << ", Next=" << z;
return 0;
}
```

Previous=99, Next=101

7.4 FUNCTIONS CALL BY VALUE

According to pass by value, when we call the function, the parameter can be passed to called function. If there any modification on parameters is occurred in called function, it will not be reflected in calling function since it is passed the actual parameter values only, not the physical location of memory.

```
void main()
{
        int k = 100;
        void fun(int k);
        cout<<endl<<"Before Calling"<<k;
        fun(k)
        cout<<"After Calling" <<k;

//called the subprogram
void fun(int k)
{
        k=200;
        cout<<endl<<"In Function "<<k;
}
```

Here, we are passing k to function from main(). In function, k is assigned with 200. But when it return back to main(), this value 200 will not taken, because it pass only the value to function. Physical memory location for k in main() and k in fun() are distinct in the sense separate memory allocations will be made.

7.5. Return by Reference

A function can also return a reference. Consider the following function.

```
int & max(int & x, int & y)
{
        If (x > y)
                return x;
```

```
}
```

SPACE FOR HINT

Since the return type of max() is int &, the function returns reference to x or y (and not the values). Then a function call such as max(a, b) will yield a reference to either a or b depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max(a,b) = -1;
```

is legal and assigns -1 to a if it is larger, otherwise -1 to b.

7.6 INLINE FUNCTIONS

The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

inline type name (arguments ...) { instructions ... }

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Example 1

```
#include <iostream.h>
const float Pi = 3.1415926;
inline float area(const float r) {return Pi * r * r;}
main()
{
  float radius;
  cout << "Enter the radius of a circle: ";
  cin >> radius;
  cout << "The area is " << area(radius) << "\n";
  return 0;
}
```

Default arguments

SPACE FOR HINT

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```
// default values in functions

#include <iostream>
int divide (int a, int b=2)
{
int r;
r=a/b;
return (r);
}
int main ()
{
cout << divide (12);
cout << endl;
cout << divide (20,4);
return 0;
}
6
5
```

As we can see in the body of the program there are two calls to function divide. In the first one:

```
divide (12)
```

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).

In the second call:

```
divide (20,4)
```

there are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

7.7. const ARGUMENTS

In C++, an argument to a function can be declared as const as shown below :

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier const tells the compiler that the fnction should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

SPACE FOR HINT

7.8 FUNCTION OVERLOADING

Overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform variety of different tasks. This is known as function polymorphism in OOP.

Using the concept of function overloading, we can design a family of function with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded add() function handles different types of data as shown below.

```
// Declarations

        int add(int a, int b);
        int add (int a, int b, int c);

//Function calls
        cout<<add(5,10);
        cout<<add(5,.10,15);


// overloaded function
#include <iostream>
int operate (int a, int b)
{
return (a*b);
}
float operate (float a, float b)
{
return (a/b);
}
int main ()
{
int x=5,y=2;
```

SPACE FOR HINT

```
float n=5.0,m=2.0;
cout << operate (x,y);
cout << "\n";
cout << operate (n,m);
cout << "\n";
return 0;
}
10
2.5
```

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

In the first call to operate the two arguments passed are of type int, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate depends on the type of the arguments passed because the function has been *overloaded.*

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

7.9 Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*.
Friends are functions or classes declared as such.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

The general syntax of the friend function is

friend return_type name (object)

```
// friend functions
#include <iostream>
```

```
class CRectangle {
int width, height;
public:
void set_values (int, int);
int area () {return (width * height);}
friend CRectangle duplicate (CRectangle);
};
void CRectangle::set_values (int a, int b) {
width = a;
height = b;
}
CRectangle duplicate (CRectangle rectparam)
{
CRectangle rectres;
rectres.width = rectparam.width*2;
rectres.height = rectparam.height*2;
return (rectres);
}
int main () {
CRectangle rect, rectb;
rect.set_values (2,3);
rectb = duplicate (rect);
cout << rectb.area();
return 0;
}
```

SPACE FOR HINT

24

The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.

USES OF FRIEND FUNCTION

There will be a situation that the non-member functions has to refer the private members. Now it is time to violate the rule seen so far that accessing private members from non-member function using the keyword friend. So friend is a special tool for accessing private members by non-members of the class. The construct of friend allows this opaque wall to be selectively broken. The case of one object sending a message

SPACE FOR HINT

to another object is common. It allows selecting outside classes or functions to access the private data of the class. With the help of this friend function the level of privacy of the data encapsulation can be reduced. One or more outside functions or an entire outside class can be declared to be a friend of a giving class. Such a friend can access the private data and functions of the given class.

Thus a friend function posses certain special characteristics:

1. The function declared as friend to the class has rights to access the private data of the lass.
2. One or more outside functions or an entire outside class can be declared to be a friend of given class.
3. A function may be declared as friend to more than one class.
4. It is not a member of the class though it is defined with in the scope of class declaration. So it cannot be called using the object of that class. It is a stand-alone function.
5. Though it has rights to access the private data, it can not refer the variables of object directly. It should refer by means of using object name for its members.
6. The friend declaration is unaffected by its location in the class. It can be declared either public or private without affecting its meaning and its access rights.
7. It has objects as arguments since it is friend to that object

7.10 Virtual Functions

A *virtual function* is a member function that is declared within a base class and redefined by a derived class.

To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism.

The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method.*

When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer.

A base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this determination is made *at run time*. Thus, when different objects

are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references.

SPACE FOR HINT

To begin, examine this short example:

```
#include <iostream>
class base
{
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base
{
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};

class derived2 : public base
{
public:
void vfunc()
{
cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

SPACE FOR HINT

As the program illustrates, inside **base**, the virtual function **vfunc()** is declared.

Notice that the keyword **virtual** precedes the rest of the function declaration. When **vfunc()** is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.)

In this program, **base** is inherited by both **derived1** and **derived2**. Inside each class definition, **vfunc()** is redefined relative to that class. Inside **main(),** four variables are declared:

Name	Type
p	base class pointer
b	object of base
d1	object of derived1
d2	object of derived2

Next, **p** is assigned the address of **b,** and **vfunc()** is called via **p**. Since **p** is pointing to an object of type **base**, that version of **vfunc()** is executed. Next, **p** is set to the address of **d1**, and again **vfunc()** is called by using **p**. This time **p** points to an object of type **derived1**. This causes **derived1::vfunc()** to be executed. Finally, **p** is assigned the address of **d2,** and **p–>vfunc()** causes the version of **vfunc()** redefined inside **derived2** to be executed. The key point here is that the kind of object to which **p** points determines which version of **vfunc()** is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism. Although you can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved. For example, assuming the preceding example, this is syntactically valid:

d2.vfunc(); // calls derived2's vfunc()
Although calling a virtual function in this manner is not wrong, it simply does not takeadvantage of the virtual nature of **vfunc()** .

C.Y.P

1. What is Function in C++?

2. What is Call by Reference ?

3. What is Function Overloading ? Explain with example.

4. What is Friend Function ?

5. What is the purpose of Virtual function and how did you declare that.

TABLE OF CONTENTS

SPACE FOR HINT

UNIT - 8

8. INTRODUCTION - Class and Objects

8.1. Structures and Classes Are Related

Structures are part of the C subset and were inherited from the C language. As you have seen, a **class** is syntactically similar to a **struct**. But the relationship between a **class** and a **struct** is closer than you may at first think. In C++, the role of the structure was expanded, making it an alternative way to specify a class. In fact, the only difference between a **class** and a **struct** is that by default all members are public in a **struct** and private in a **class**. In all other respects, structures and classes are equivalent.

That is, in C++, a *structure defines a class type*. For example, consider this short program,which uses a structure to declare a class that controls access to a string:

```
// Using a structure to define a class.

#include <iostream>
#include <cstring>
struct mystr
{
void buildstr(char *s); // public
void showstr();
private: // now go private
char str[255];
} ;
void mystr::buildstr(char *s)
{
if(!*s) *str = '\0'; // initialize string
else strcat(str, s);
}
void mystr::showstr()
{
cout << str << "\n";
}
int main()
{
mystr s;
s.buildstr(""); // init
s.buildstr("Hello ");
s.buildstr("there!");
s.showstr();
return 0;
}
```

This program displays the string **Hello there!**.

The class **mystr** could be rewritten by using **class** as shown here:

SPACE FOR HINT

```
class mystr {
char str[255];
public:
void buildstr(char *s); // public
void showstr();
} ;
```

You might wonder why C++ contains the two virtually equivalent keywords **struct** and **class**. This seeming redundancy is justified for several reasons. First, there is no fundamental reason not to increase the capabilities of a structure. In C, structures already provide a means of grouping data. Therefore, it is a small step to allow them to include member functions. Second, because structures and classes are related, it may be easier to port existing C programs to C++. Finally, although **struct** and **class** are virtually equivalent today, providing two different keywords allows the definition of a **class** to be free to evolve. In order for C++ to remain compatible with C, the definition of **struct** must always be tied to its C definition.

Although you can use a **struct** where you use a **class**, most programmers don't. Usually it is best to use a **class** when you want a class, and a **struct** when you want a
C-like structure.

8.2 CLASS

A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

A class is a user defined data type that holds both the data and functions. It consists of two kinds of properties namely data and functions. The data of a class is called member data or **data member** and the functions are called **member functions** that operate on data of a class. The member data of a class will be normally be hidden to outside of the class. But accessing member data can be routed through its ***member functions*** so as to ensure the protection mechanism of the class. The variables or instance of the class is called object.

SPACE FOR HINT

Class : **ITEM**
DATA Number Cost .....
METHODS Getdata Putdata .....

Representation of class

8.3 OBJECT

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. As pointed out earlier, objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C. When a program is executed, the objects interact by sending messages to one another. Each objects contain data and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted and the type of response returned by the objects.

The general form of a simple **class** declaration is

class *class-name*
{
private data and functions
public:
public data and functions
} ***object name list*;**

Of course, the *object name list* may be empty.

8.4 Creating Objects

Remember that the declaration of item as shown above does not define any objects of item but only specifies what they will contain. Once a

class has been declared, we can create variable of that type using the class name. For example,

SPACE FOR HINT

item x; // memory for x is created

creates a variable x of type item. In C++ the class variables are known as objects. Therfore, x is called an object of type item. We may also declare more than one object in one statement. Example

item x,y,z;

8.5 Accessing Class Members

The private data of a class can be accessed only through the member functions of that class. The main() cannot contain statement that access number and cost direcly. The following is the format for calling a member function.

Object-name.function-nam(actual-arguments)

For example the function call statement
x.getdata(100,75.5);

is valid and assigns the value 100 to number and 75.5 to cost of the object x by implementing the getdata() function.

8.6. DEFINING MEMBER FUNCTION

Member function can be defined in two places :

- Outside- the class definition
- Inside the class definition

Outside- the class definition

A important definition between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which class the function belongs to. The general form of a member function definition is

Return-type ***class-name*** *:: function-name(argument declaration)*
{
Function body
}

The membership label class-name:: tells the compiler that the function function name belongs to the class class-name. That is, the scope of the

SPACE FOR HINT

function is restricted to the class-name specified in the header line. The symbol :: is called the *scope resolution operator.*

Inside the class definition

Another method of defining member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows

```
class item
{

        int number;
        float cost;
public:
        void getdata(int a, float b);
        //inline function
        void putdata(void);
        {
                cout<<number<<;
                cout<<cost<<;
        }
};
```

CLASS SPECIFICATION

Generally, a class specification has two par...

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

```
class  class name
{
    private:
        variable declarations;
        function declarations;
    public:
        variable declarations;
        function declarations;
};
```

The class declarations is similar to a struct declaration. The keyword class specifies that what follows is an abstract data of type class name. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called members. They are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public. The keywords private and public are known as visibility labels. Note that these keywords are followed by a colon.

SPACE FOR HINT

The members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from within the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are private. If both the labels are missing, then, by default, all the members are private. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as data members and the functions are known as member functions. Only the member functions can have access to the private data members and private functions. However, the public members(both functions and data) can be accessed from outside the class. The binding of data and functions together into a single class-type variable is referred to as encapsulation.

8.6.1 A c++ program with class

```
class CRectangle
{
int x, y;
public:
void set_values (int,int);
int area (void);
} rect;
```

Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship int and a have in the following declaration:

int a;

SPACE FOR HINT

where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

Here is the complete example of class CRectangle:

```
// classes example
#include <iostream>
class CRectangle
{
int x, y;
public:
void set_values (int,int);
int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b)
{
x = a;
y = b;
}
int main ()
{
CRectangle rect;
rect.set_values (3,4);
cout << "area: " << rect.area();
return 0;
}
area: 12
```

The most important new thing in this code is the operator of scope (::, two colons) included in the definition of set_values(). It is used to define a member of a class from outside the class definition itself.

You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class

declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

SPACE FOR HINT

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it. For example, following with the previous example of class CRectangle, we could have declared the object rectb inaddition to the object rect:

```
// example: one class, two objects
#include <iostream>
class CRectangle
{
int x, y;
public:
void set_values (int,int);
int area () {return (x*y);
}
};
void CRectangle::set_values (int a, int b)
{
x = a;
y = b;
```

SPACE FOR HINT

```
}
int main ()
{
CRectangle rect, rectb;
rect.set_values (3,4);
rectb.set_values (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
rect area: 12
rectb area: 30
```

In this concrete case, the class (type of the objects) to which we are talking about is CRectangle, of which there are two instances or objects: rect and rectb. Each one of them has its own member variables and member functions.

Notice that the call to rect.area() does not give the same result as the call to rectb.area(). This is because each object of class CRectangle has its own variables x and y, as they, in some way, have also their own function members set_value() and area() that each uses its object's own variables to operate.

That is the basic concept of *object-oriented programming*: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members. Notice that we have not had to give any parameters in any of the calls to rect.area or rectb.area. Those member functions directly used the data members of their respective objects rect and rectb.

8.7. NESTING OF MEMBER FUNCTOIN

A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

8.8. PRIVATE MEMBER FUNCTION

A private member function can only be called by another function that is member of its class. Even an object cannot invoke a private function using the dot operator.

```
class example
```

```
        void update(void);
        void write(void);
};
```

If s1 is an object of sample then

```
s1.read();      //won't work; objects cannot access private members
```

Is illegal. However the function read() can be called by the function update() to update the value of m

```
void sample::update(void)
{
        read();         // simple call; no object used
}
```

Static Data Members and Static Member function

Both function and data members of a class can be made **static**. This section explains the consequences of each.

8.9 Static Data Members

When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a **static** member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable. All **static** variables are initialized to zero before the first object is created.

When you declare a **static** data member within a class, you are *not* defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the **static** variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated. (Remember, a class declaration is simply a logical construct that does not have physical reality.)

To understand the usage and effect of a **static** data member, consider this program:

```
#include <iostream>
class shared {
static int a;
int b;
public:
void set(int i, int j) {a=i; b=j;}
void show();
```

SPACE FOR HINT

SPACE FOR HINT

```
} ;
int shared::a; // define a
void shared::show()
{
cout << "This is static a: " << a;
cout << "\nThis is non-static b: " << b;
cout << "\n";
}
int main()
{
shared x, y;
x.set(1, 1); // set a to 1
x.show();
y.set(2, 2); // change a to 2
y.show();
x.show(); /* Here, a has been changed for both x and y
because a is shared by both objects. */
return 0;
}
```

This program displays the following output when run.

```
This is static a: 1
This is non-static b: 1

This is static a: 2
This is non-static b: 2

This is static a: 2
This is non-static b: 1
```

Notice that the integer **a** is declared both inside **shared** and outside of it. As mentioned earlier, this is necessary because the declaration of **a** inside **shared** does not allocate storage.

A **static** member variable exists *before* any object of its class is created. For example, in the following short program, **a** is both **public** and **static**. Thus it maybe directly accessed in **main().** Further, since **a** exists before an object of **shared** is created, **a** can be given a value at any time. As this program illustrates, the value of **a** is unchanged by the creation of object **x**. For this reason, both output statements display the same value: 99.

```
#include <iostream>
class shared
{
public:
static int a;
} ;
int shared::a; // define a
```

```
int main()
{
// initialize a before creating any objects
shared::a = 99;
cout << "This is initial value of a: " << shared::a;
cout << "\n";
shared x;
cout << "This is x.a: " << x.a;
return 0;
}
```

Notice how **a** is referred to through the use of the class name and the scope resolution
operator. In general, to refer to a **static** member independently of an object, you must
qualify it by using the name of the class of which it is a member.

8.10 Static Member Functions

Member functions may also be declared as **static**. There are several restrictions placed on **static** member functions. They may only directly refer to other **static** members of the class. (Of course, global functions and data may be accessed by **static** member functions.)

A **static** member function does not have a **this** pointer. There cannot be a **static** and a non-**static** version of the same function. A **static** member function may not be virtual. Finally, they cannot be declared as **const** or **volatile**.

Example

```
#include <iostream>
class cl
{
static int rescurce;
public:
static int get_resource();
void free_resource() { resource = 0; }
};
int cl::resource; // define resource
int cl::get_resource()
{
if(resource) return 0; // resource already in use
else {
resource = 1;
return 1; // resource allocated to this object
}
}
int main()
{
```

SPACE FOR HINT

```
cl ob1, ob2;
/* get_resource() is static so may be called independent
of any object. */
if(cl::get_resource()) cout << "ob1 has resource\n";
if(!cl::get_resource()) cout << "ob2 denied resource\n";
ob1.free_resource();
if(ob2.get_resource()) // can still call using object syntax
cout << "ob2 can now use resource\n";
return 0;
}
```

Actually, **static** member functions have limited applications, but one good use for them is to "preinitialize" private **static** data before any object is actually created. For example, this is a perfectly valid C++ program:

```
#include <iostream>
using namespace std;
class static_type {
static int i;
public:
static void init(int x) {i = x;}
void show() {cout << i;}
};
int static_type::i; // define i
int main()
{
// init static data before object creation
static_type::init(100);
static_type x;
x.show(); // displays 100
return 0;
}
```

8.11 ARRAYS OF OBEJCTS

An array can be of any data type including struct. Similarly, we can also have arrays of variables that are of the type class. Such variables are called arrays of objects. Consider the following class definition.

```
class employee
{
        char name[30];
        float age;
        public:
                void getdata(void);
                void putdata(void);
};
```

SPACE FOR HINT

The identified employee is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example.

```
employee manager[3];
```

The array manager contains three objects (managers), namely, manager[0], manager[1], manager[2], of type employee class. Since any sparray of object behaves like any other array, we can use the usual array-accessing methods to access individual elements and then the dot member operator to access the member functions.

An array of objects is stored inside the memory in the same way as a multi-dimension array. Only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

```
#include <iostream.h>

class employee
{
        char name[30]
        float age;
        public:
                void getdata(void);
                void putdata(void);
};
void employee :: getdata (void)
{
                cout<<"Enter Name:";
                cin>>name;
                cout<<"Enter age:";
                cin>>age;
}
void employee :: putdata (void)
{
                cout<<"Name:"<<name<<"\n";
                cout<<"Age:"<<age<<"\n";
}
const  int  size=3;
main()
{
        employee manager[size];
        for (int i=0;i<size;i++)
        {
                cout<<" \nDetails of manager"<<i+1<<"\n";
                manager[i].getdata();
        }
                cout<<"\n";
        for (int i=0;i<size;i++)
        {
```

SPACE FOR HINT

```
                cout<<" \nManager"<<i+1<<"\n";
                manager[i].putdata();
        }
}
```

Interactive input

```
Details of manager1
Enter name : xxxx
Enter age :    45

Details of manager2
Enter name : yyyy
Enter age :    36

Details of manager3
Enter name : zzzz
Enter age :    50
```

Program output

```
Manager1
Name : xxxx
Age:  45

Manager2
Name : yyyy
Age:  36

Manager3
Name : zzzz
Age:  50
```

8.12 OBJECTS AS FUNCTION ARGUMENTS

Like any other data type, an object may be used as a function argument. This can be done in two ways.

- A copy of the entire object is passed to the function .
- Only the address of the object is transferred to the function.

The first method is called pass-by-value. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called *pass-by-reference.* When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by-reference method

is more efficient since it requires to pass only the address of the object and not the entire object.

SPACE FOR HINT

```
#include <iostream.h>
class time
{
        int hours;
        int minutes;
        pubic:
            void gettime (int h, int m)
            {   hours =h; minutes=m;}
            void puttime(void)
            {
                cout<<hours<<"hours and ";
            cout<<minutes<<" minutes " <<"\n";
            }
            void sum(time, time)
};

void time :: sum(time t1, time t2)
{
    minutes = t1.minutes + t2.minutes;
    hours = minutes/60;
    minutes = minutes %60;
    hours = hours + t1.hours + t2.hours
}

main()
{
    time t1,t2,t3;
    t1.gettime(2,45);
    t2.gettime(3,30);

    t3.sum(t1,t2);

    cout<<"T1= "; t1.puttime();
    cout<<"T2= "; t2.puttime();
    cout<<"T3= "; t3.puttime();
}
```

OUTPUT

T1 = 2 hours and 45 minutes
T2 = 3 hours and 60 minutes
T3 = 6 hours and 15 minutes

SPACE FOR HINT

8.13 FRIENDLY FUNCTIONS

There will be a situation that the non-member functions has to refer the private members. Now it is time to violate the rule seen so far that accessing private members from non-member function using the keyword friend. So friend is a special tool for accessing private members by non-members of the class. The construct of friend allows this opaque wall to be selectively broken. The case of one subject sending a message to another object is common. It allows selecting outside classes or functions to access the private data of the class. With the help of this friend function the level of privacy of the data encapsulation can be reduced. One or more outside functions or an entire outside class can be declared to be a friend of a giving class. Such a friend can access the private data and functions of the given class.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the class as shown below

```
class ABC
{
.....
.....
public:
.....
.....
friend void xyz(void);   //declaration
};
```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal c++ function. The function definition does not use either the keyword friend or the scope operator **::**. The function that are declared with the keyword friend are known as *friend* function.

Thus a friend function posses certain special characteristics:

- The function declared as friend to the class has rights to access the private data of the class.
- One or more outside functions or an entire outside class can be declared to be a friend of a given class.
- A function may be declared as friend to more than one class.
- It is not a member of the class though it is defined with in the scope of class declaration. So it cannot be called using the object of that class. It is a stand-alone function.
- Though it has rights to access the private data, it can not refer the variables of object directly. It should refer by means of using object name for its members.
- The friend declaration is unaffected by its location in the class. It can be declared either public or private without affecting its meaning and its access rights.
- It has objects as arguments since it is friend to that object

SPACE FOR HINT

Example : Friend function

```
#include<iostream.h>
class sample
{
        int a;
        int b;
        public:
        void setvalue()
        {
                a=25, b = 40;
                friend float mean (sample s)   // friend declared
};

float mean(sample s)
{
        feturn float (s.a + s.b) / 2.0
}

main()
{
        sample X;
        X.setvalue();
        Cout<< "Mean value =  " <<mean(X)<<"\n";
}
```

Output

Mean value : 32.5

8.14 RETURNING OBJECTS

A function may return an object to the caller. For example, this is a valid C++ program:

```
// Returning objects from a function.
#include <iostream>
class myclass
{
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass f(); // return object of type myclass
int main()
{
myclass o;
o = f();
```

SPACE FOR HINT

```
cout << o.get_i() << "\n";
return 0;
}
myclass f()
{
myclass x;
x.set_i(1);
return x;
}
```

When an object is returned by a function, a temporary object is automatically created that holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object may cause unexpected side effects in some situations.

For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it.

8.1.1 Introduction - Constructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function area() before having called function set_values()? Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class,and cannot have any return type; not even void.

We are going to implement CRectangle including a constructor:

```
// example: class constructor
#include <iostream>
class CRectangle
{
int width, height;
public:
CRectangle (int,int);
int area () {return (width*height);}
};
CRectangle::CRectangle (int a, int b)
{
width = a;
height = b;
```

```
}
int main ()
{
CRectangle rect (3,4);
CRectangle rectb (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
rect area: 12
rectb area: 30
```

SPACE FOR HINT

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set_values(), and have included instead a constructor that performs a similar action: it initializes the values of x and y with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

CRectangle rect (3,4);

CRectangle rectb (5,6);

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

You can also see how neither the constructor prototype declaration (within the class) nor the latter constructordefinition include a return value; not even void.

MULTIPLE CONSTRUCTORS IN A CLASS

8.1.2 Parameterized Constructors

It is possible to pass arguments to constructor functions. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object. For example, here is a simple class that includes a parameterized constructor:

```
#include <iostream>
class myclass
{
int a, b;
public:
myclass(int i, int j) {a=i; b=j;}
void show()
```

SPACE FOR HINT

```
{cout << a << " " << b;}
};
int main()
{
myclass ob(3, 5);
ob.show();
return 0;
}
```

Notice that in the definition of **myclass(),** the parameters **i** and **j** are used to give initial values to **a** and **b**.

The program illustrates the most common way to specify arguments when you declare an object that uses a parameterized constructor function. Specifically, this statement

```
myclass ob(3, 4);
```

causes an object called **ob** to be created and passes the arguments **3** and **4** to the **i** and **j** parameters of **myclass().** You may also pass arguments using this type of declaration
statement:

```
myclass ob = myclass(3, 4);
```

However, the first method is the one generally used.

Actually, there is a small technical difference between the two types of declarations that relates to copy constructors. (Copy constructors are discussed in Chapter 14.)
Here is another example that uses a parameterized constructor function. It creates a

8.1.3 Multiple Constructors In A Class : Overloaded Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

```
// overloading class constructors
#include <iostream>
class CRectangle
{
int width, height;
public:
```

```
CRectangle ();
CRectangle (int,int);
int area (void) {return (width*height);}
};
CRectangle::CRectangle ()
{
width = 5;
height = 5;
}
CRectangle::CRectangle (int a, int b)
{
width = a;
height = b;
}
int main ()
{
CRectangle rect (3,4);
CRectangle rectb;
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
rect area: 12
rectb area: 25
```

SPACE FOR HINT

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both width and height with a value of 5.

```
CRectangle rectb; // right
CRectangle rectb(); // wrong!
```

8.1.4 Default Constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

```
class CExample
{
public:
int a,b,c;
void multiply (int n, int m) { a=n; b=m; c=a*b; };
};
```

The compiler assumes that CExample has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
CExample ex;
```

SPACE FOR HINT

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

```
class CExample
{
public:
int a,b,c;
CExample (int n, int m) { a=n; b=m; };
void multiply () { c=a*b; };
};
```

Here we have declared a constructor that takes two parameters of type int. Therefore the following object declaration would be correct:

```
CExample ex (2,3);
```

But,

```
CExample ex;
```

Would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor. But the compiler not only creates a default constructor for you if you do not specify your own. It provides three special member functions in total that are implicitly declared if you do not declare your own. These are the *copy constructor*, the *copy assignment operator*, and the default destructor.

The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object. For CExample, the copy constructor implicitly declared by the compiler would be something similar to:

```
CExample::CExample (const CExample& rv)
{
a=rv.a; b=rv.b; c=rv.c;
}
```

Therefore, the two following object declarations would be correct:

```
CExample ex (2,3);

CExample ex2 (ex); // copy constructor (data copied from ex)
```

8.1.1 Copy Constructors

One of the more important forms of an overloaded constructor is the ***copy constructor***. Defining a copy constructor can help you prevent problems that might occur when one object is used to initialize another.

SPACE FOR HINT

One of the most common is when an object allocates memory when it is created. For example, assume a class called *MyClass* that allocates memory for each object when it is created, and an object *A* of that class. This means that *A* has already allocated its memory.

Further, assume that *A* is used to initialize *B*, as shown here:

MyClass B= A;

If a bitwise copy is performed, then *B* will be an exact copy of *A*. This means that *B* will be using the same piece of allocated memory that *A* is using, instead of allocating its own. Clearly, this is not the desired outcome.

For example, if *MyClass* includes a destructor that frees the memory, then the same piece of memory will be freed twice when *A* and *B* are destroyed!

The same type of problem can occur in two additional ways: first, when a copy of an object is made when it is passed as an argument to a function; second, when a temporary object is created as a return value from a function. Remember, temporary objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances.

To solve the type of problem just described, C++ allows you to create a ***copy constructor***, which the compiler uses when one object initializes another. When a copy constructor exists, the default, bitwise copy is bypassed. The most common general form of a copy constructor is

classname (const *classname &o*)

```
{
// body of constructor
}
```

Here, *o* is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them. However, in all cases the first parameter must be a reference to the object doing the initializing.

It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first is assignment. The second is initialization, which can occur any of three ways:

SPACE FOR HINT

- When one object explicitly initializes another, such as in a declaration
- When a copy of an object is made to be passed to a function
- When a temporary object is generated (most commonly, as a return value)

The copy constructor applies only to initializations. For example, assuming a class called **myclass**, and that **y** is an object of type **myclass**, each of the following statements involves initialization.

myclass x = y; // y explicitly initializing x

func(y); // y passed as a parameter

y = func(); // y receiving a temporary, return object

example : copy constructors

```
#include<iostream.h>
class code
{
        int id;
public :
        code(){{
        code(int a) [id =a;];
        code(code &x)
        {
                id=x.id;
        }
        void display (void)
        {
                cout<<id;
        };
main()
{
        code A(100);
        code B(A);
        code C=A;
        code D;
        D=A;

        cout<<"\n id of A :   "; A.display();
        cout<<"\n id of B :   "; B.display();
        cout<<"\n id of C :   "; C.display();
        cout<<"\n id of D :   "; D.display();
```

Output

SPACE FOR HINT

id of A: 100
id of B: 100
id of C: 100
id of D: 100

8.1.6 Dynamic Constructor

The constructor can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory of each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is knows as dynamic construction of objects. The memory is allocated with the help of the new operator.

8.1.7 Destructors

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceeded by a tilde. For example, the destructor for the class integer can be defined as below :

```
~integer()
```

As destructor never takes any argument nor does it return any value. It will be involved implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare the destructors in a program since it release memory space for future use.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

```
// example on constructors and destructors
#include <iostream>
class CRectangle
{
int *width, *height;
public:
CRectangle (int,int);
~CRectangle ();
int area () {return (*width * *height);}
};
```

SPACE FOR HINT

```
CRectangle::CRectangle (int a, int b)
{
width = new int;
height = new int;
*width = a;
*height = b;
}
CRectangle::~CRectangle ()
{
delete width;
delete height;
}
int main ()
{
CRectangle rect (3,4), rectb (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
rect area: 12
rectb area: 30
```

8.2.1 INTRODUCTION - Polymorphism

Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation.

A real-world example of polymorphism is a thermostat. No matter what type of furnace your house has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (method) you have. For example, if you want a 70-degree temperature, you set the thermostat to 70 degrees. It doesn't matter what type of furnace actually provides the heat.

This same principle can also apply to programming. For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, **push()** and **pop()** , that can be used for all three stacks. In your program you will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being stored.

Thus, the interface to a stack—the functions **push()** and **pop()** —are the same no matter which type of stack is being used. The individual

versions of these functions define the specific implementations (methods) for each type of data.

SPACE FOR HINT

Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the *specific action* (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the *general interface*.

The first object-oriented programming languages were interpreters, so polymorphism
was, of course, supported at run time. However, C++ is a compiled language. Therefore, in C++, both run-time and compile-time polymorphism are supported.

8.2.2 Pointers to Objects

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow(–>) operator instead of the dot operator. The next program illustrates how to access an object given a pointer to it:

```
#include <iostream>
using namespace std;

class cl
 {
int i;
public:
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob(88), *p;
p = &ob; // get address of ob
cout << p->get_i(); // use -> to call get_i()
return 0;
}
```

As you know, when a pointer is incremented, it points to the next element of its type. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the base type of the pointer. (That is, it is relative to the type of data that the pointer is declared as pointing to.) The same is true of pointers to objects. For example, this program uses a pointer to access all three elements of array
ob after being assigned **ob**'s starting address:

```
#include <iostream>
class cl
```

SPACE FOR HINT

```
{
int i;
public:
cl() { i=0; }
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob[3] = {1, 2, 3};
cl *p;
int i;
p = ob; // get start of array
for(i=0; i<3; i++) {
cout << p->get_i() << "\n";
p++; // point to next object
}
return 0;
}
```

You can assign the address of a public member of an object to a pointer and then access that member by using the pointer. For example, this is a valid C++ program that displays the number **1** on the screen:

```
#include <iostream>
class cl {
public:
int i;
cl(int j) { i=j; }
};
int main()
{
cl ob(1);
int *p;
p = &ob.i; // get address of ob.i
cout << *p; // access ob.i via p
return 0;
}
```

Because **p** is pointing to an integer, it is declared as an integer pointer. It is irrelevant
that **i** is a member of **ob** in this situation.

8.2.3 The this POINTER

C++ uses a unique keyword called this to represent an object that invokes a member function. this is a pointer that points to the object for which this function was called. For example, the function call A.max() will set the pointer this to the address of the object A. The starting

address is the same as the address of the first variable in the class structure.

SPACE FOR HINT

This unique pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions. Consider the following simple example:

```
class ABC
{
        int a;
        ....
        ....
};
```

The private variable a can be used directly inside a member function, like

```
a=123;
```

We can also use the following statement to do the same job:

```
this->a=123;
```

Since C++ permits the use of shorthand form a=123, we have not been using the pointer this explicitly so far. However, we have been implicitly using the pointer this when overloading the operators using member function.

8.2.4 Pointers to Derived Types

In general, a pointer of one type cannot point to an object of a different type. However, there is an important exception to this rule that relates only to derived classes. To begin, assume two classes called **B** and **D**. Further, assume that **D** is derived from the base class **B**. In this situation, a pointer of type **B *** may also point to an object of type **D**. More generally, a base class pointer can also be used as a pointer to an object of any class derived from that base.

Although a base class pointer can be used to point to a derived object, the opposite is not true. A pointer of type **D *** may not point to an object of type **B**. Further, although you can use a base pointer to point to a derived object, you can access only the members of the derived type that were imported from the base. That is, you won't be able to access any members added by the derived class. (You can cast a base pointer into a derived pointer and gain full access to the entire derived class, however.)

Here is a short program that illustrates the use of a base pointer to access

derived objects.

```
#include <iostream>
```

SPACE FOR HINT

```
class base {
int i;
public:
void set_i(int num) { i=num; }
int get_i() { return i; }
};
class derived: public base {
int j;
public:
void set_j(int num) { j=num; }
int get_j() { return j; }
};
int main()
{
base *bp;
derived d;
bp = &d; // base pointer points to derived object
// access derived object using base pointer
bp->set_i(10);
cout << bp->get_i() << " ";

/* The following won't work. You can't access element of
a derived class using a base class pointer.
bp->set_j(88); // error
cout << bp->get_j(); // error
*/
return 0;
}
```

As you can see, a base pointer is used to access an object of a derived class. Although you must be careful, it is possible to cast a base pointer into a pointer of the derived type to access a member of the derived class through the base pointer.
For example, this is valid C++ code:

```
// access now allowed because of cast
((derived *)bp)->set_j(88);
cout << ((derived *)bp)->get_j();
```

It is important to remember that pointer arithmetic is relative to the base type of the pointer. For this reason, when a base pointer is pointing to a derived object, incrementing the pointer does not cause it to point to the next object of the derived type. Instead, it will point to what it thinks is the next object of the base type. This, of course, usually spells trouble. For example, this program, while syntactically correct, contains this error.

```
// This program contains an error.
#include <iostream>
class base {
```

```
int i;
public:
void set_i(int num) { i=num; }
int get_i() { return i; }
};
class derived: public base {
int j;
public:
void set_j(int num) {j=num;}
int get_j() {return j;}
};
int main()
{
base *bp;
derived d[2];
bp = d;
d[0].set_i(1);
d[1].set_i(2);
cout << bp->get_i() << " ";
bp++; // relative to base, not derived
cout << bp->get_i(); // garbage value displayed
return 0;
}
```

The use of base pointers to derived types is most useful when creating run-time polymorphism through the mechanism of virtual functions

SPACE FOR HINT

8.2.5 Pointers to Class Members

C++ allows you to generate a special type of pointer that "points" generically to a
member of a class, not to a specific instance of that member in an object. This sort of
pointer is called a *pointer to a class member* or a *pointer-to-member*, for short. A pointer to a member is not the same as a normal C++ pointer. Instead, a pointer to a member provides only an offset into an object of the member's class at which that member can be found. Since member pointers are not true pointers, the . and -> cannot be applied to them. To access a member of a class given a pointer to it, you must use the special pointer-to-member operators .* and –>*. Their job is to allow you to access a member of a class given a pointer to that member.

Here is an example:

```
#include <iostream>
class cl {
public:
cl(int i) { val=i; }
int val;
int double_val() { return val+val; }
```

SPACE FOR HINT

```
};
int main()
{
int cl::*data; // data member pointer
int (cl::*func)(); // function member pointer
cl ob1(1), ob2(2); // create objects
data = &cl::val; // get offset of val
func = &cl::double_val; // get offset of double_val()
cout << "Here are values: ";
cout << ob1.*data << " " << ob2.*data << "\n";
cout << "Here they are doubled: ";
cout << (ob1.*func)() << " ";
cout << (ob2.*func)() << "\n";
return 0;
}
```

8.2.6 Virtual Functions

A *virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to
the derived class. That is, the redefinition creates a *specific method.*

When accessed "normally," virtual functions behave just like any other type of class
member function. However, what makes virtual functions important and capable of
supporting run-time polymorphism is how they behave when accessed via a pointer.

A base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this determination is made *at run time*. Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references. To begin, examine this short example:

```
#include <iostream>
class base {
public:
virtual void vfunc() {
```

SPACE FOR HINT

```
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {

class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

As the program illustrates, inside **base**, the virtual function **vfunc()** is declared. Notice that the keyword **virtual** precedes the rest of the function declaration. When **vfunc()** is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed.(However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.)

In this program, **base** is inherited by both **derived1** and **derived2**. Inside each class definition, **vfunc()** is redefined relative to that class. Inside **main()**, four variables are declared:

Name	Type
p	base class pointer
b	object of base
d1	object of derived1

SPACE FOR HINT

d2 object of derived2

Next, **p** is assigned the address of **b,** and **vfunc()** is called via **p**. Since **p** is pointing to an object of type **base**, that version of **vfunc()** is executed. Next, **p** is set to the address of **d1**, and again **vfunc()** is called by using **p**. This time **p** points to an object of type **derived1**. This causes **derived1::vfunc()** to be executed. Finally, **p** is assigned the address of **d2,** and **p->vfunc()** causes the version of **vfunc()** redefined inside **derived2** to be executed. The key point here is that the kind of object to which **p** points determines which version of **vfunc()** is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism.

Although you can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved. For example, assuming the preceding example, this is syntactically valid:

```
d2.vfunc(); // calls derived2's vfunc()
```

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc()** .

At first glance, the redefinition of a virtual function by a derived class appears similar to function overloading. However, this is not the case, and the term *overloading* is not applied to virtual function redefinition because several differences exist. Perhaps the most important is that the prototype for a redefined virtual function must match exactly the prototype specified in the base class. This differs from overloading a normal function, in which return types and the number and type of parameters may differ. (In fact, when you overload a function, either the number or the type of the parameters *must* differ! It is through these differences that C++ can select the correct version of an overloaded function.) However, when a virtual function is redefined, all aspects of its prototype must be the same. If you change the prototype when you attempt to redefine a virtualfunction, the function will simply be considered overloaded by the C++ compiler, and its virtual nature will be lost.

Another important restriction is that virtual functions must be nonstatic members of the classes of which they are part.

They cannot be **friends.**

Finally, constructor functions cannot be virtual, but destructor functions can.

Because of the restrictions and differences between function overloading and virtual function redefinition, the term *overriding* is used to describe virtual function redefinition by a derived class.

SPACE FOR HINT

8.2.7 Pure Virtual Functions

When a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created.

Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.

A *pure virtual function* is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form:

virtual ***type func-name(parameter-list)*** **= 0;**

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

The following program contains a simple example of a pure virtual function.

The base class, **number**, contains an integer called **val**, the function **setval()** , and the pure virtual function **show()** . The derived classes **hextype**, **dectype**, and **octtype** inherit **number** and redefine **show()** so that it outputs the value of **val** in each respective number base (that is, hexadecimal, decimal, or octal).

```
#include <iostream>
class number
{
protected:
int val;
public:
void setval(int i) { val = i; }
// show() is a pure virtual function
virtual void show() = 0;
};
class hextype : public number {
public:
void show() {
cout << hex << val << "\n";
```

SPACE FOR HINT

```
};
class dectype : public number {
public:
void show() {
cout << val << "\n";
}
};
class octtype : public number {
public:
void show() {
cout << oct << val << "\n";
}
};
int main()
{
dectype d;
hextype h;
octtype o;
d.setval(20);
d.show(); // displays 20 - decimal
h.setval(20);
h.show(); // displays 14 - hexadecimal
o.setval(20);
o.show(); // displays 24 - octal
return 0;
}
```

Although this example is quite simple, it illustrates how a base class may not be able to meaningfully define a virtual function. In this case, **number** simply provides the common interface for the derived types to use. There is no reason to define **show()** inside **number** since the base of the number is undefined.

Of course, you can always create a placeholder definition of a virtual function. However, making **show()** pure also ensures that all derived classes will indeed redefine it to meet their own needs. Keep in mind that when a virtual function is declared as pure, all derived classes must override it. If a derived class fails to do this, a compile-time error will result.

Example 2

```
// virtual members
#include <iostream>
class CPolygon
{
protected:
```

```
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b; }
virtual int area ()
{ return (0); }
};
class CRectangle: public CPolygon
{
public:
int area ()
{ return (width * height); }
};
class CTriangle: public CPolygon
{
public:
int area ()
{ return (width * height / 2); }
};

int main ()
{
CRectangle rect;
CTriangle trgl;
CPolygon poly;
CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
CPolygon * ppoly3 = &poly;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly3->set_values (4,5);
cout << ppoly1->area() << endl;
cout << ppoly2->area() << endl;
cout << ppoly3->area() << endl;
```

SPACE FOR HINT

SPACE FOR HINT

```
return 0;
}
20
10
0
```

Now the three classes (CPolygon, CRectangle and CTriangle) have all the same members: width, height, set_values() and area().

The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. You can verify if you want that if you remove this virtual keyword from the declaration of area() within CPolygon, and then you run the program the result will be 0 for the three polygons instead of 20, 10 and 0. That is because instead of calling the corresponding area() function for each object (CRectangle::area(),CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called in all cases since the calls are via a pointer whose type is CPolygon*.

Therefore, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the above example.

C.Y.P

1. What is Class ?
2. What is Object ?
3. Write a program which implements the Class and Object concepts
4. What is Constructors ? Explain its purpose.
5. What is Destructors ?

TABLE OF CONTENTS

SPACE FOR HINT

UNIT - 9

9.1 INTRODUCTION
OPERATOR OVERLOADING AND TYPE CONVERSIONS

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. We have stated more than once that C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as Operator overloading.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can almost create a new language of our own by the creative use of the function and operator overloading techniques. We can overload (give additional meaning to) all the C++ operators except the following:

- Class member access operators (.,.*).
- Scope resolution operator (::).
- Size operator (sizeof).
- Conditional operator (?:).

The excluded operators are very few when compared to the large number of operators which qualify for the operator overloading definition.

Although the semantics of an operator can be extended, we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associativity For example, the operator will enjoy higher precedence than the addition operator.

Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

9.2 DEFINING OPERATOR OVERLOADING

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is

done with the help of a special function, called *operator function,* which describes the task. The general form of an operator function is:

SPACE FOR HINT

```
returntype classname :: operator op(arg-list)
{
Function body      //task defined
}
```

Where return type is the type of value returned by the specified operation and op is the operator being overload. The op is preceded by the keyword operator. Operator op is the function name.

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions. Arguments may be passed either by value or by reference.

Operator functions are declared in the class using prototype as follows:

```
Vector operator + (vector);        //vector addition
Vector operator – ();              //unary minus
Friend vector operator + (vector, vector);   //vector addition
Friend vector operator-(vector);    //unary minus
Vector operator- (vector & a);    //subtraction
int operator == (vector, vector)    //comparison
Friend int operator == (vector, vector)    comparison
```

Vector is a data type of class and may represent both magnitude and direction (as in physics and engineering) or a series points called elements (as in mathematics).

The process of overloading involves the following steps:

1. First, create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required operators.

SPACE FOR HINT

Overloaded operator functions can be invoked by expressions such as op x or x op

for unary operators and
x op y

For binary operators.

Op x (or x op) would be interpreted as

operator op(x)

for **friend** functions. Similarly, the expression x op y would be interpreted as either

x.operator op(y)
in case of member functions, or
operator op (x, y)

in case of **friend** functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

9.3 OVERLOADING UNARY OPERATORS

Let us consider the unary minus operator. A minus operator, when used as unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variable. The unary minus when applied to an object should change the sign of each of its data items.
How the unary minus operator is overloaded

```
//////////[ OVERLOADING UNARY MINUS ]//////////
#include <iostream.h>
class space
{
int x;
int y;
int z;
public:
void getdata(int a, int b, int c);
void display(void);
void operator-();          //overload unary minus
```

```
};
void space : : getdata (int a, int b, int c)
{
x=a;
t=b;
z=c;
}
void space : : display(void)
{
cout << x << " ";
cout << y << " ";
cout << z << "\n";
}
void space : : operator-()          //Defining operator-()
{
x = -x;
y = -y;
z = -z;
}
main()
{
space s;
s.getdata(10, -20, 30);
cout << "S : ";
s.display();
-s;                                 //activities operator-()
cout << "S  : ";
s.display();
}
```

SPACE FOR HINT

The program produces the following output:

```
S : 10 -20 30
S :-10 20 -30
```

Note that the function operator-() takes no argument. Then, what does this operator function do ? It changes the sign of data members of the object S. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Remember, a statement like

```
S2=-S1;
```

SPACE FOR HINT

Will not work because, the function operator-() does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
Friend void operator- (space & s);      //declaration
void operator-(space & s)               //definition
{
s.x = -s.x;
s.y =-s.y;
s.z =-s.z;
}
```

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-(). Therefore, the changes made inside the operator function will not reflect in the called object.

9.4 OVERLOADING BINARY OPERATORS

We have just seen how to overload a unary operator. The Same mechanism can be used to overload a binary operator. In chapter 6, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C=sum(A, B));     //functional notation
```

was used. The functional notation can be replaced by a natural-looking expression

```
C = A+B;          //arithmetic notation
```

By overloading the + operator using an **operator+()** function.

```
//////////[ OVERLOADING + OPERATOR ]//////////
#include<iostream.h>

class complex
{
float x;                          //real part

float y;                          //imaginary part
public:
complex()  {  }                       //constructor1
complex(float real, float imag)     //constructor2
```

```
        { x = real;  y = imag;}
complex operator+(complex);
void display(void);
};
complex complex : : operator+ (complex c)
{
complex  temp;        //temporary
temp.x = x+ c.x;       //float addition
temp.y = y + c.y;      //float addition
return(temp);
}
void complex :: display(void)
{
        cout<< x  << " + j" << y  << "\n":
}
main()
{
complex C1,  c2,  C3;      //invokes constructor 1
C1 = complex(2.5, 3.5);   //invokes constructor2
C2 = complex(1.6, 2.7);    //invokes constructor2
C3 = c1 + c2;                  //invokes operator+()

cout  <<  "C1  =  ";  C1.display();
cout  <<  "C2  =  ";  C2.display();
cout  <<  "C3  =  ";  C3.display();
}
```

The output of program:

```
C1  =  2.5  +  j3.5
C2  =  1.6  +  j2.7
C3  =  4.1  +  j6.2
```

9.5 MANIPULATION OF STRING USING OPERATORS

/////[MATHEMATICAL OPERATIONS ON STRINGS]/////

```
#include <string.h>
#include <iostream.h>
{
char *p;
int len;
```

SPACE FOR HINT

SPACE FOR HINT

```
public:
string() {len = 0; p = 0;}   //create null string
string(const char * s)    //create string from array s
string(const string & s);  //copy constructor
~string() {delete p;}        //destructor
friend string operator+(const string & s, const string &t);
friend int operator<=(const string & s, const string & t);
friend void show(const string s);
};
string : : string(const char * s)
{
len=strlen(s);
p=new char[len + 1];
strcpy(p,s);
}
string : : string(const string & s)
{
len = s.len;
p=new char[len + 1];
strcpy(p, s.p);
}
//.....OVERLOADING + OPERATOR.....
string operator+(const string & s, const string & t)
{
string temp;
temp.len=s.len + t.len;
temp.p=new char[temp.len + 1];
strcpy(temp.p,s.p);
strcat(temp.p,t.p);
return(temp);
}
//.....OVERLOADING<=OPERATOR.....
int operator<=(const string & s,const string & t)
{
int m = strlen(s.p);
int n = strlen(t.p);
if(m <= n)    return(10;
else return(0);
}
void show (const string s)
{
cout << s.p;
}
main()
```

```
{
string s1 = "New";
string s2 ="York";
string s3 ="Delhi";
string t1,t2,t3,t4;

t1 = s1;
t2 = s2;
t3 = s1+s2;
t4 = s1+s3;

cout << "\nt1 = "; show(t1);
cout << "\nt2 = "; show(t2);
cout << "\n";
cout << "\nt3 = "; show(t3);
cout << "\nt4 = "; show(t4);
cout << "\n\n";
if(t3 <= t4)
{
show(t3);
cout << " smaller than ";
show(t4);
cout << "\n";
}
else
{
    show(t4);
cout << " smaller than  ";
show(t3);
cout << "\n";
}
}
```

SPACE FOR HINT

The following is the output of program

```
t1=New
t2=York
t3=New York
t4=New Delhi
New York smaller than New Delhi
```

SPACE FOR HINT

RULES FOR OVERLOADING OPERATORS

Its simple to redefine the operators, there are certain restriction and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is user-defined type.
3. We cannot change the basic meaning of an operator. That is, we cannot redefine plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. That cannot be overridden.
5. There are some operators than cannot be overloaded
6. We cannot use friend functions to overload certain operators. However, member function can be used to overload them.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return to explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +,-,* and / must explicitly return a value. They must not attempt to change their own arguments.

Any operators in C++ can be overloaded **except** the following operators.

- :: - Scope resolution operator
- .,.* - Class members access operator
- sizeof - size of operator
- ?: - conditional operator
- . - membership operator

9.7 TYPE CONVERSIONS

We know that constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left.

For example, the statements

```
int m;
```

```
float x=3.14159;
m=x;
```

SPACE FOR HINT

Convert x to an integer before its value is assigned to m thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types. What happens when they are user-defined data types?

Consider the following statement that adds two objects and then assigns the result to a third object.

```
V3 = v1 + v2;  //v1, v2 and V3 are class type objects
```

When the objects are of the same class type, the operators of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, the values of all the data member of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable? Or, what if they belong to two different classes?

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. We must, therefore, design the conversion routines by ourselves, if such operation are required

Three types of situations might arise in the data conversions between uncompatible types:

1. Conversion from built-in type to class type.
2. Conversion from class type to built-in type.
3. Conversion from one class type to another class type.

We shall discuss all the three cases in detail.

Basic To Class Type

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. These are all examples where constructors perform a defector type conversion from the argument's type to the constructor's class type.

SPACE FOR HINT

Consider the following constructor:

```
string :: string(char *a)
{
length = strlen(a);
p = new char[length + 1];
strcpy (P,a);
}
```

This constructor builds a string type object from a char * type variable a. the variables length class, it can be used for conversion from char * type to string type. Example:

```
string s1, s2;
char * name1= "IBM PC";
char * name2= "apple computers";
s1= string(name1);
s2= name2;
```

The statement

s1=string(name1);

First converts **name1** from **char** * type to string type and then assigns the string type values to the object s1. The statement

s2=name2;

also does the same job by invoking the constructor implicitly.

Let us consider another example of converting an int type to a class type.

```
class time
{
int    hrs;
int    mins;

public:
. . . . . .
. . . . . .
time(int t)          //constructor

        {
hours = t / 60;     //t in minutes
mins  = t % 60;
        }
};
```

SPACE FOR HINT

The following conversion statements can be used in a function:

```
time = T1;        //object T1 created
int duration = 85;
t1 = duration;    // int to class type
```

After this conversion, the hrs member of TI will contain a value of 1 and mins member a value of 25, denoting 1 hour and 25 minutes.

Note that the constructors used for the type conversion take a single argument whose type is to be converted.

In both the examples, the left-hand operand of = operator is always a class object. Therefore, we can also accomplish this conversion using an overloaded = operator.

Class to Basic type

The constructors did a fine job in type conversion from a basic to class type. What about the conversion from a class to basic type? The constructor functions do not support this operation.

Luckily , C++ allows us to define a overloaded casting operator that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function, usually referred to as a conversion function. Is:

```
operator typename()
{
        .....
        ..... (Function statement)
        .....
}
```

This function converts class type data to type name. For example, the operator double() converts a class object to type double, the operator int() converts a class type object type object to type int,and so on.

Consider the following conversion function:

```
vector :: operator double()
{
double sum = 0;
for(int I = 0; <size; i++)
```

SPACE FOR HINT

```
sum = sum + v[i] * v[i];
return sqrt(sum);
}
```

This function converts a vector to the corresponding scalar magnitude. Recall that the magnitude of a vector is given by the square root of the sum of the squares of its components. The operator double() can be used as follows:

```
double length = double(V1);
```

or

```
double length = V1;
```

where V1 is an object of type vector. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Since it is a member function, it is invoked by the object and therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

In the string example described in the previous section, we can do the conversion from string to char * as follows:

```
string :: operator char*()
{
    return(p);
}
```

9.7 EXCEPTION HANDLING

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

SPACE FOR HINT

A exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

```
// exceptions
#include <iostream>
int main ()
{
try
{
throw 20;
}
catch (int e)
{
cout << "An exception occurred. "
cout << "Exception Nr. " << e << endl;
}
return 0;
}
```

An exception occurred. Exception Nr. 20

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the catch keyword. As you can see, it follows immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught. We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

SPACE FOR HINT

```
try {
      // code here
    }

catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char.
After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.
It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
try {
try {
// code here
}
catch (int n)
{
throw;
}
}
catch (...)
{
cout << "Exception occurred";
}
}
```

Exception specifications

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

```
float myfunction (char param) throw (int);
```

This declares a function called myfunction which takes one agument of type char and returns an element of type float. The only exception that this function might throw is an exception of type int. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular int-type handler.

If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

```
int myfunction (int param) throw(); // no exceptions allowed
```

int myfunction (int param); // all exceptions allowed

SPACE FOR HINT

Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called exception and is defined in the <exception> header file under the namespace std. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called what that returns a null-terminated character sequence (char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

```
// standard exceptions
#include <iostream>
#include <exception>
class myexception: public exception
{
virtual const char* what() const throw()
{
return "My exception happened";
}
} myex;
int main ()
{
try
{
throw myex;
}
catch (exception& e)
{
cout << e.what() << endl;
}
return 0;
}
```

My exception happened.

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of class myexception. All exceptions thrown by components of the C++ Standard library throw exceptions derived from this std::exception class.

These are:

Exception	**description**
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid
ios_base::failure	thrown by functions in the iostream library

For example, if we use the operator new and the memory cannot be allocated, an exception of type bad_alloc is thrown:

```
try
```

SPACE FOR HINT

```
{
int * myarray= new int[1000];
}
catch (bad_alloc&)
{
cout << "Error allocating memory." << endl;
}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. If you want to force a bad_alloc exception to see it in action, you can try to allocate a huge array; On my system, trying to allocate 1 billion ints threw a bad_alloc exception. Because bad_alloc is derived from the standard base class exception, we can handle that same exception by catching references to the exception class:

```
// bad_alloc standard exception
#include <iostream>
#include <exception>
int main ()
{
try
{
int* myarray= new int[1000];
}
catch (exception& e)
{
cout << "Standard exception: " << e.what()
<< endl;
}
return 0;
}
```

C.Y.P

1. What is Operator Overloading ?
2. Explain how to overload the Binary Operator with example.
3. What are the various rules to Overload the Operators ?
4. What are the various operators that cannot be Overloaded ?
5. What is Type Conversion.
6. What is Exception Handling ? Explain with example

Table of Contents

SPACE FOR HINT

UNIT – 10

10.1 INTRODUCTION - INHERITANCE

A key feature of C++ classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own.

Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.

In keeping with standard C++ terminology, a class that is inherited is referred to as a *base class*. The class that does the inheriting is called the *derived class*. Further, a derived class can be used as a base class for another derived class. In this way, multiple inheritance is achieved C++'s support of inheritance is both rich and flexible.

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by simply separating the different base classes with commas in the derived class declaration.

10.2 DEFINING DERIVED CLASSES

This could be represented in the world of classes with a class CPolygon from which we would derive the two other ones: CRectangle and CTriangle.

The class CPolygon would contain members that are common for both types of polygon. In our case: width and height. And CRectangle and CTriangle would be its derived classes, with specific features that are different from one type of polygon to the other.Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B. In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

When a class inherits another, the members of the base class become members of the
derived class. Class inheritance uses this general form:

class *derived-class-name : access base-class-name*
{
.....//
..... // members of derived class

```
        ...../
};
```

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers protected and private. This access specifier describes the minimum access level for the members that are inherited from the base class.

SPACE FOR HINT

The access status of the base-class members inside the derived class is determined by *access*. The base-class access specifier must be either **public, private,** or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier. Let's examine the ramifications of using **public** or **private** access.

When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.

We can summarize the different access types according to who can access them in the following way:

Access	**public**	**protected**	**private**
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	noS

Where "not members" represent any access from outside the class, such as from main(), from another class or from a function.

In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:

```
CPolygon::width // protected access
CRectangle::width // protected access
CPolygon::set_values() // public access
CRectangle::set_values() // public access
```

This is because we have used the public keyword to define the inheritance relationship on each of the derived classes:

```
class CRectangle: public CPolygon { ... }
```

SPACE FOR HINT

This public keyword after the colon (:) denotes the maximum access level for all the members inherited from the class that follows it (in this case CPolygon). Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

If we specify a more restrictive access level like protected, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: private, all the base class members are inherited as private.

For example, if daughter was a class derived from mother that we defined as:

class daughter: protected mother;

This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother.

If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with class keyword and public for those declared with struct.

What is inherited from the base class ?

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

For example:

SPACE FOR HINT

```
// constructors and derived classes

#include <iostream>
class mother
{
public:
mother ()
{ cout << "mother: no parameters\n"; }

mother (int a)
{ cout << "mother: int parameter\n"; }
};
class daughter : public mother
{
public:
daughter (int a)
{ cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
public:
son (int a) : mother (a)
{ cout << "son: int parameter\n\n"; }
};
int main ()
{
daughter cynthia (0);
son daniel(0);
return 0;
}
```

```
mother: no parameters
daughter: int parameter
mother: int parameter
son: int parameter
```

Notice the difference between which mother's constructor is called when a new daughter object is created and which when it is a son object. The difference is because the constructor declaration of daughter and son: daughter (int a) // nothing specified: call default son (int a) : mother (a) // constructor specified: call this

Types of Inheritance

- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

SPACE FOR HINT

A derived class with only one base class is called **single inheritance.**

A derived class with several base classes is called **multiple inheritance.**

The mechanism of deriving a class from another derived class is known as **multilevel inheritance.**

The traits of one class may be inherited by more than one class is known as **hierarchical inheritance.**

Various forms of inheritance that could be used for writing extensible programs.

10.3 SINGLE INHERTANCE

Let us consider a simple example to illustrate inheritance. The following program shows a base class **B** and a derived class **D**. The class **B** contains one private data member, one public data member, and three public member functions. The class **D** contains one private data member and two public member functions.

Example : single inheritance

```
#include<iostream.h>
class B
{
        int a;          // private ; not inheritable
        public:
        int b;          // public ; ready for inheritance
        void get_ab();
        int get_a(void);
        void show_a(void);
};
class D : public B      // public derivation
{
        int c
        public:
        void mul(void);
        void display(void);

};
//  functions defined //
void B::get_ab(void)
{
        a=5; b=10;
}
int B :: get_a()
{
```

```
        return a;
}
void B :: show_a()
{
        cout<< "a=  "<<a<<"\n";
}
void D :: mul()
{
        c= b * get_a();
}
void D :: display()
{
cout<< "a = " <<get_a()<<"\n";
cout<< "b = " << b <<"\n";
cout<< "c = " << c <<"\n\n";

}
// MAIN PROGRAM
main()
{
        D d;

        d.get_ab();
        d.mul();
        d.show_a();
        d.display();

        d.b = 20;
        d.mul();
        d.display();
}
```

Output

```
        a = 5
        a = 5
        b = 10
        c = 50

        a = 5
        b = 20
        c = 100
```

Example : single inheritance - private

```
#include<iostream.h>
class B
{
        int a;          // private ; not inheritable
```

SPACE FOR HINT

SPACE FOR HINT

```
        public:
        int b;          // public ; ready for inheritance
        void get_ab();
        int get_a(void);
        void show_a(void);
};
class D : private B     // public derivation
{
        int c
        public:
        void mul(void);
        void display(void);

};
//  functions defined //
void B::get_ab(void)
{
        cout<<"Enter values for a and b: ";
        cin>> a >> b;
}
int B :: get_a()
{
        return a;
}
void B :: show_a()
{
        cout<< "a=  "<<a<<"\n";
}
void D :: mul()
{
        get_ab();
        c= b * get_a();         // 'a' cannot be used directly
}
void D :: display()
{
Show_a();       //outputs value of 'a'
cout<< "b = " << b <<"\n";
cout<< "c = " << c <<"\n\n";

}
// MAIN PROGRAM
main()
{
        D d;

        d.mul();                //d.get_ab();  WON'T WORK
        d.display();            //d.show_a();  WON'T WORK

        //d.b=20;               WON'T WORK;  b has become private
```

```
        d.mul();
        d.display();
}
```

SPACE FOR HINT

Output

```
Enter value for a and b : 5 10
a = 5
b = 10
c = 50

Enter value for a and b : 12 20
a = 12
b = 20
c = 240
```

10.4 Multiple Inheritance

A class can inherit the attributes of two or more class is known as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.

The syntax of a derived class with multiple base classes is as follows:

```
Class D:visibility B-1, visibility B-2,....
{
        .....
        ..... (Body of D)
        .....
};
```

Where visibility may be either public or private. The base classes are separated by commas.

For example, if we had a specific class to print on screen (COutput) and we wanted our classes CRectangle and CTriangle to also inherit its members in addition to those of CPolygon we could write:

```
class CRectangle: public CPolygon, public COutput;

class CTriangle: public CPolygon, public COutput;
```

here is the complete example:

```
// multiple inheritance
#include <iostream>
```

SPACE FOR HINT

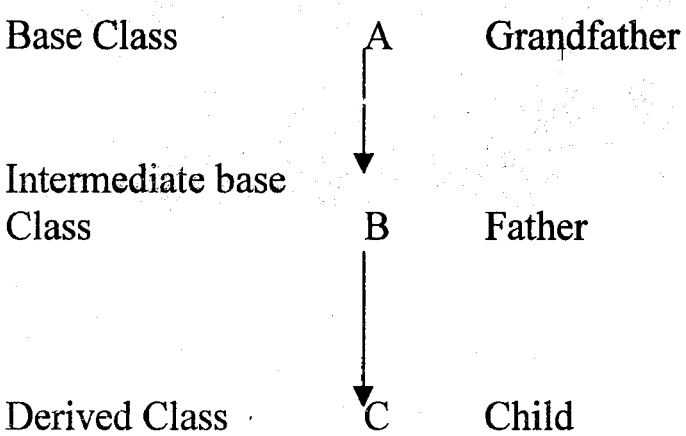
```
class CPolygon
{
protected:
int width, height;
public:
void set_values (int a, int b)
{ width=a; height=b;}
};

class COutput
{
public:
void output (int i);
};
void COutput::output (int i)
{
cout << i << endl;
}
class CRectangle: public CPolygon, public COutput
{
public:
int area ()
{ return (width * height); }
};

class CTriangle: public CPolygon, public COutput
{
public:
int area ()
{ return (width * height / 2); }
};
int main ()
{
CRectangle rect;
CTriangle trgl;
rect.set_values (4,5);
trgl.set_values (4,5);
rect.output (rect.area());
trgl.output (trgl.area());
return 0;
}
20
10
```

10.5 Multilevel Inheritance

It is uncommon that a class is derived from another derived class. The class serves as a base class for the derived class B which in turn serves as base class for the derived class C. The class B is knows as

intermediate class since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path

SPACE FOR HINT



A derived class with multilevel inheritance is declared as follows :

```
        class A(.....)                  //base class
class B:public A(....._          //B derived from A
class C:public B(....._          //C derived from B
```

This process can be extended to any number of levels.

Example : Multilevel Inheritance

```
#include<iostream.h>
class student
{
        protected:
                int roll_number;
        public:
                void get_number(int a);
                void put_number(void);
};

void student ::  get_number(int a)
{
        roll_number = a;
}

void student:: put_number(void)
{
        cout<<"Roll No:" << roll_number <<"\n";

}

class test : public student                \\ FIRST LEVEL DERIVATION
{
        protected:
                float sub1;
```

SPACE FOR HINT

```
        float sub2;
    public:
        void get_marks(float x, float y);
void put_marks(void);
};

void test :: get_marks(float x, float y)
{
    sub1 = x;  sub2=y;
}

void test ::put_marks()
{
    cout<<"Marks in SUB1" <<sub1  <<"\n";
    cout<<"Marks in SUB2" <<sub2  <<"\n";

}

class result : public test          \\      SECOND      LEVEL
DERIVATION
{
    float total;
    public:
        void display(void);
};

void result :: display (void)
{
    total = sub1 + sub2;
    put_number();
    put_marks();
    cout<< "Total Score : " << total <<"\n";
}

main()
{
    result student_1;
    student_1.get_number(111);
    student_1.get_marks(75.0,59.5);
    student_1.display();
}
```

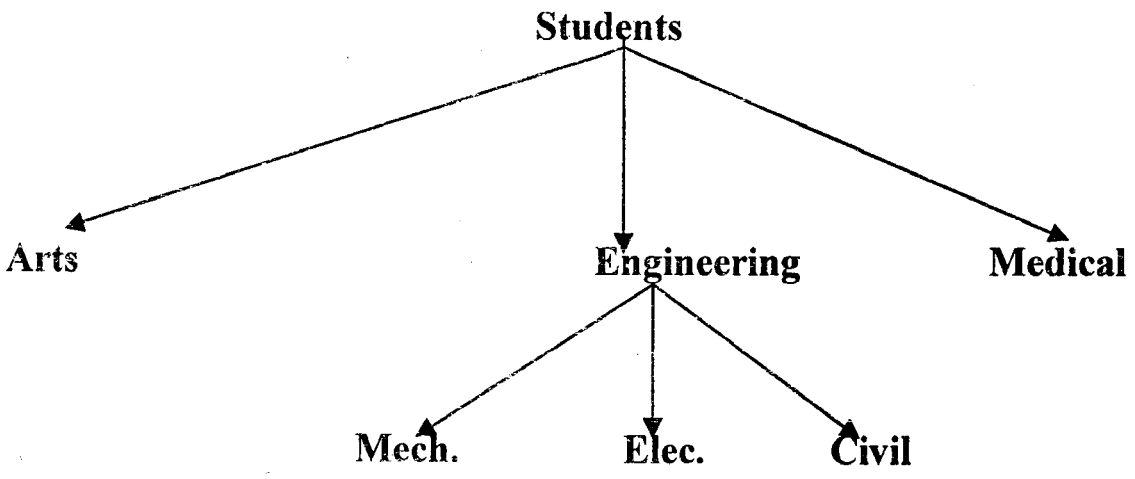
Output

```
    Roll Number : 111

    Marks in SUB1 = 75

    Marks in SUB2 = 59.5

    Total = 134.5
```

10.6 HIERARCHICAL INHERITANCE

SPACE FOR HINT

Here the base class will include all the features that are common to the subclasses. A subclass can be considered by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.



Hierarchical class of students

Example Program : Hierarchical Inheritance

Here is a program illustrating inheritance. It creates two derived classes of **building**
using inheritance; one is **house**, the other, **school**.
member

```
#include <iostream>
class building {
int rooms;
int floors;
int area;
public:
void set_rooms(int num);
int get_rooms();
void set_floors(int num);
int get_floors();
void set_area(int num);
int get_area();
};
// house is derived from building
class house : public building {
int bedrooms;
int baths;
public:
void set_bedrooms(int num);
int get_bedrooms();
void set_baths(int num);
int get_baths();
```

SPACE FOR HINT

```
};
// school is also derived from building
class school : public building {
int classrooms;
int offices;
public:
void set_classrooms(int num);
int get_classrooms();
void set_offices(int num);
int get_offices();
};
void building::set_rooms(int num)
{
rooms = num;
}
void building::set_floors(int num)
{
floors = num;
}
void building::set_area(int num)
{
area = num;
}
int building::get_rooms()
{
return rooms;
}
int building::get_floors()
{
return floors;
}
int building::get_area()
{
return area;
}
void house::set_bedrooms(int num)
{
bedrooms = num;
}
void house::set_baths(int num)
{
baths = num;
}
int house::get_bedrooms()
{
return bedrooms;
}
int house::get_baths()
```

```
}
void school::set_classrooms(int num)
{
classrooms = num;
}
void school::set_offices(int num)
{
offices = num;
}
int school::get_classrooms()
{
return classrooms;
}
int school::get_offices()
{
return offices;
}
int main()
{
house h;
school s;
h.set_rooms(12);
h.set_floors(3);
h.set_area(4500);
h.set_bedrooms(5);
h.set_baths(3);
cout << "house has " << h.get_bedrooms();
cout << " bedrooms\n";
s.set_rooms(200);
s.set_classrooms(180);
s.set_offices(5);
s.set_area(25000);
cout << "school has " << s.get_classrooms();
cout << " classrooms\n";
cout << "Its area is " << s.get_area();
return 0;
}
```

SPACE FOR HINT

The output produced by this program is shown here.

```
house has 5 bedrooms
school has 180 classrooms
Its area is 25000
```

As this program shows, the major advantage of inheritance is that you can create a general classification that can be incorporated into more specific ones. In this way, each object can precisely represent its own subclass.

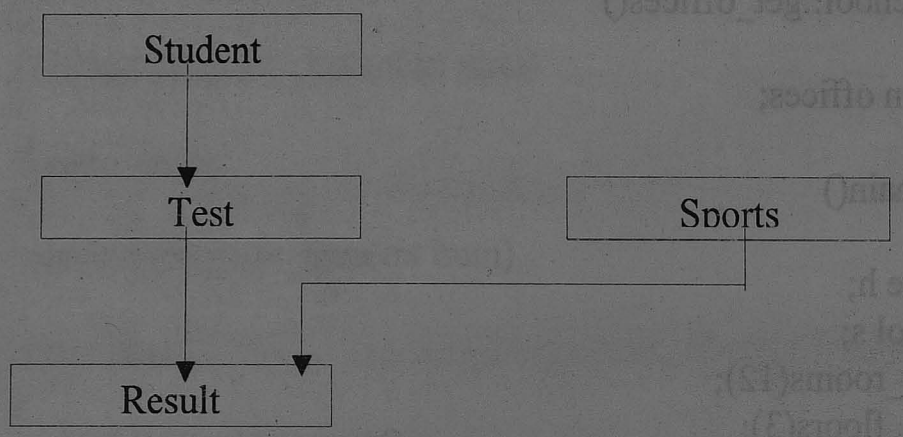
SPACE FOR HINT

When writing about C++, the terms *base* and *derived* are generally used to describe the inheritance relationship. However, the terms *parent* and *child* are also used. You may also see the terms *superclass* and *subclass*.

Aside from providing the advantages of hierarchical classification, inheritance also provides support for run-time polymorphism through the mechanism of **virtual** functions.

10.7 HYBRID INHERITANCE

There could be a situation where we need to apply two or more types of inheritance to design a program. Assume that we have to give weightage for ***sports*** before finalizing the results. The weightage for sports is stored in a separate class called ***sports.***



The sports class might look like:

```
class sports
{
        protected:
                float score;
        public:
                void get_score(float);
                void put_score(void);
};
```

The result will have both the multilevel and multiple inheritance and its declaration would be as follows:

```
class results : public test, public sports
{
        ........
        ........
};
```

where test itself is a derived class from student. That is

```
class test : public student
{
```

SPACE FOR HINT

```
};
```

Example

```
#include<iostream.h>
class student
{
        protected:
                int roll_number;
        public:
                void get_number(int a)
                {
                        roll_number = a;
                }

                void put_number(void)
                {
                        cout<<"Roll No:" << roll_number <<"\n";
                }
};

class test : public student
{
        protected:
                float part1, part2;
        public:
                void get_marks(float x, float y)
                {       part1=x; part2=y;}

void put_marks(void)
                {
                        cout<<"Marks obtained: " <<"\n"

                        cout<<"Part1<<part1 <<"\n"
                        cout<<"Part2<<part2 <<"\n";
                }
};

class sports
{
        protected:
                float score;
        public:
                void get_score(float s)
                {       score =s; }
                void put_score(void)
                {cout<<" Sports wt: "<<score<<"\n\n"; }
};
```

SPACE FOR HINT

```
class result : public test, public sports
{
        float total;
        public:
                void display(void);
};

void result :: display (void)
{
        total = part1 + part2 + score;
        put_number();
        put_marks();
        put_score();
        cout<< "Total Score : " << total <<"\n";
}

main()
{
        result student_1;
        student_1.get_number(1234);
        student_1.get_marks(27.5,33.0);
        student_1.get_score(6.0);
        student_1.display();
}
```

Output

```
Roll No : 1234
Marks obtained:
Part1 = 27.5
Part2 = 33
Sports wt:6

Total score: 66.5
```

10.8 Templates – Introduction

SPACE FOR HINT

A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data. As you probably know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quicksort sorting algorithm is the same whether it is applied to an array of integers or an array of floats.

It is just that the type of the data being sorted is different. By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself.

A generic function is created using the keyword **template**. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed

10.9 Function templates – Introduction

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type. The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;

template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way. For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
```

SPACE FOR HINT

```
return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this stillundefined type.

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

```
// function template
#include <iostream>
template <class T>
T GetMax (T a, T b)
{
T result;
result = (a>b)? a : b;
return (result);
}
int main ()
{
int i=5, j=6, k;
long l=10, m=5, n;
k=GetMax<int>(i,j);
n=GetMax<long>(l,m);
cout << k << endl;
cout << n << endl;
return 0;
}
6
10
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like. In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

SPACE FOR HINT

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type: T result;

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j;

GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II
#include <iostream>
using namespace std;
template <class T>
T GetMax (T a, T b) {
return (a>b?a:b);
}
int main () {
int i=5, j=6, k;
long l=10, m=5, n;
k=GetMax(i,j);
n=GetMax(l,m);
cout << k << endl;
cout << n << endl;
return 0;
}
6
10
```

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

SPACE FOR HINT

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;
long l;
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b) {
return (a<b?a:b);
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
or simply:
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

10.10 Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
T values [2];
public:
mypair (T first, T second)
{
values[0]=first; values[1]=second;
}
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:
mypair<int> myobject (115, 36);

this same class would also be used to create an object to store any other type:

mypair<double> myfloats (3.0, 2.18);

The only member function in the previous class template has been defined inline within the class declaration itself.

In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

```
// class templates
#include <iostream>
using namespace std;
template <class T>
class mypair {
T a, b;
public:
mypair (T first, T second)
{a=first; b=second;}
T getmax ();
};
template <class T>
T mypair<T>::getmax ()
{
T retval;
retval = a>b? a : b;
return retval;
}
int main () {
mypair <int> myobject (100, 75);
cout << myobject.getmax();
return 0;
}
100
```

Notice the syntax of the definition of member function getmax:

```
template <class T>
T mypair<T>::getmax ()
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type

SPACE FOR HINT

SPACE FOR HINT

returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.

For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type:

```
// template specialization
#include <iostream>
using namespace std;
// class template:
template <class T>
class mycontainer {
T element;
public:
mycontainer (T arg) {element=arg;}
T increase () {return ++element;}
};
// class template specialization:
template <>
class mycontainer <char> {
char element;
public:
mycontainer (char arg) {element=arg;}
char uppercase ()
{
if ((element>='a')&&(element<='z'))
element+='A'-'a';
return element;
}
};
int main () {
mycontainer<int> myint (7);
mycontainer<char> mychar ('j');
cout << myint.increase() << endl;
cout << mychar.uppercase() << endl;
return 0;
}
8
```

J

This is the syntax used in the class template specialization:

template <> class mycontainer <char> { ... };

First of all, notice that we precede the class template name with an emptytemplate<> parameter list. This is to explicitly declare it as a template specialization. But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:

template <class T> class mycontainer { ... };

template <> class mycontainer <char> { ... };

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

Non-type parameters for templates

Besides the template arguments that are preceded by the class or typename keywords , which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:

```
// sequence template
#include <iostream>
using namespace std;
template <class T, int N>
class mysequence {
T memblock [N];
public:
void setmember (int x, T value);
T getmember (int x);
};
template <class T, int N>
void mysequence<T,N>::setmember (int x, T value)
{
memblock[x]=value;
}
template <class T, int N>
T mysequence<T,N>::getmember (int x) {
```

SPACE FOR HINT

SPACE FOR HINT

```
return memblock[x];
}
int main () {
mysequence <int,5> myints;
mysequence <double,5> myfloats;
myints.setmember (0,100);
myfloats.setmember (3,3.1416);
cout << myints.getmember(0) << '\n';
cout << myfloats.getmember(3) << '\n';
return 0;
}
100
3.1416
```

It is also possible to set default values or types for class template parameters. For example, if the previous class template definition had been:

template <class T=char, int N=10> class mysequence {..};
We could create objects using the default template parameters by declaring: mysequence<> myseq; Which would be equivalent to:

mysequence<char,10> myseq;

Since no code is generated until a template is instantiated when required, compilers are prepared to allow the inclusion more than once of the same template file with both declarations and definitions in a project without generating linkage errors.

C.Y.P

What is Inheritance ?

What is the advantage of Inheritance ?

Explain the various types of Inheritance with example program.

What is Template ?

What is Class Template ?

EXAMPLE PROGRAM

Inline Function

```
#include<iostream.h>
#include<conio.h>
inline int simple(int p,int n,int i)
{
return(p*n*i/(float)100);
}
void main()
{
clrscr();
int p,n,i;
cout<<"\t simple interest\n";
cout<<"*-------------------*";
cout<<"\n enter principle amount:";
cin>>p;
cout<<"\n enter number of years:";
cin>>n;
cout<<"\n enter interest:";
cin>>i;
cout<<"\n simple interest :"<<simple(p,n,i);
getch();
}
```

Output :

enter principle amount:5000

enter number of years:2

enter interest:2

simple interest :200

Default Argument

```
#include<iostream.h>
#include<conio.h>
float power(float,int n=2);
void main()
{
clrscr();
float p,m;
int n;
cout<<"\t Default Argument";
cout<<"to calculate 'm' to the power'n'";
```

```
cout<<"\n\t-------";
cout<<"\n enter the value of 'm':";
cin>>m;
cout<<"\n enter the value of 'n':";
cin>>n;
p=power(m,n);
cout<<"\n power value for "<<m<<"to thepower"<<n<<"\t",p;
cout<<"\n\n enter the value of 'm':";
cin>>m;
p=power(m),
cout<<"\n\npower value for "<<m<<"to the power 2 is"<<p;
getch();
}
float power(float m,int n)
{
float r=1;
for(int i=1;i<=n;i++)
r=r*m;
return r;
}
```

Output:

```
Default Argument
---------------------
to calculate 'm' to the power'n'
      -------
 enter the value of 'm':3

 enter the value of 'n':3

 power value for 3to thepower3  27

 enter the value of 'm':4

power value for 4to the power 2 is16
```

Implementation of Function Overloading

```
#include<iostream.h>
#include<conio.h>
int volume(int);
double volume(double,int);
long volume(long,int,int);
int main()
{
clrscr();
cout<<"volume of cube:"<<volume(10)<<"\n";
cout<<"volume of circle:"<<(5.5,8)<<"\n";
```

```
cout<<"volume of rectangle:"<<(10L,75,10)<<"\n";
getch();
return 0;
}
int volume(int s)
{
return(s*s*s);
}
double volume(double r,int h)
{
return(3.14*r*h);
}
long volume(long l,int b,int h)
{
return(l*b*h);
}
```

Output:

volume of cube:1000
volume of circle:8
volume of rectangle:10

Implementation of Friend Function

```
#include<iostream.h>
#include<conio.h>
class second;
class first
{
private:
int a;
public:
void agetno();
friend int maxno(first x,second y);
};
void first::agetno()
{
cout<<"enter the no:";
cin>>a;
}
class second
{
private :
int b;
public:
void bgetno();
friend int maxno(first x,second y)
};
```

```
void second::bgetno()
{
cout<<"enter number:";
cin>>b;
}
int maxno(first x,second y)
{
if (x.a>y.b)
return(x.a);
else
return(y.b);
}

int main()
{
first f1;
second s1;
int big;
clrscr();
f1.agetno();
s1.bgetno();
big=maxno(f1,s1);
cout<<"big is:"<<big;
getch();
return 0;
}
```

Output:

```
enter the no:56
enter number:45
big is:56
```

Class & Object Example program

```
#include<iostream.h>
#include<conio.h>
class stud
{
int rno;
char name[20];
int m[4];
public:
void getdata();
void display();
};
void stud::getdata()
```

```
int i;
cout<<"roll no:";
cin>>rno;
cout<<endl<<"name:";
cin>>name;
cout<<endl<<"marks:"<<endl;
for(i=1;i<=3;i++)
{
cout<<"marks"<<i<<":";
cin>>m[i];
}
}
void stud::display()
{
int i,tot=0;
cout<<"\n";
cout<<rno<<"\t"<<name<<"\t\t";
for(i=1;i<=3;i++)
{
cout<<""<<m[i]<<"\t";
tot=tot+m[i];
}
cout<<tot;
}

void main()
{
int n,j,i;
stud s[10];
clrscr();
cout<<"enter no of stud:";
cin>>n;
for(j=1;j<=n;j++)
s[j].getdata();
cout<<"roll no \t name \t mark1 \t mark2 \t mark3 \t total \n";
for(i=1;i<=n;i++)
s[i].display();
getch();
}
```

Output

```
roll no:345

name:lakshmi

marks:
marks1:56
marks2:67
marks3:78
```

roll no	name	mark1	mark2	mark3	total
123	rama	89	78	56	223
345	lakshmi	56	67	78	201

Example for Constructor and Destructor

```
#include<iostream.h>
#include<conio.h>
class integer
{
int m,n;
public:
integer()
{
m=0;
n=0;
}
integer (int a,int b)
{
m=a;
n=b;
}
integer (integer &i)
{
m=i.m;
n=i.n;
}
void display()
{
cout<<"m="<<m<<endl;
cout<<"n="<<n<<endl;
}
~integer()
{  }
};
void main()
{
clrscr();
integer i1;
integer i2(20,40);
integer i3(i2);
i2.display();
i3.display();
getch();
}
```

```
m=20
n=40
m=20
n=40
```

Unary Operator Overloading

```
#include<iostream.h>
#include<conio.h>
class space
{
int x;
int y;
int z;
public:
void getdata(int a,int b,int c);
void display(void);
void operator-();
};
void space::getdata(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
void space::display()
{
cout<<x<<"\t"<<y<<"\t"<<z;
}
void space::operator-()
{
x=-x;
y=-y;
z=-z;
}
int main()
{
space s;
clrscr();
s.getdata(10,-20,30);
cout<<"s:";
s.display();
-s;
cout<<"\n"<<"s:";
s.display();
getch();
return 0;
}
```

Output:

```
s:10 -20    30
s:-10  20    -30
```

Class and Object Concept – Bank Transaction

```
#include<iostream.h>
#include<iomanip.h>
#include<string.h>
#include<conio.h>
class bank
{
private:
char accname[20];
char acctype;
int accno;
float balance;
public:
bank(char tname[],char ttype, int taccno,float tbalance)
{
strcpy(accname,tname);
acctype=ttype;
accno=taccno;
balance=tbalance;
}
bank(){}
void deposit(float sum)
{
balance+=sum;
}
void withdraw(float sum)
{
if(balance>sum)
balance-=sum;
else
cout<<"not enough fund";
}
void display()
{
cout<<"\ncurrent balance";
cout<<"\n name:"<<accname;
cout<<"\n balance";
cout<<setprecision(2)<<balance<<endl;
}
};
```

```
void main()
{
bank bank1("Rama",'s',1001,100);
float amt;
int choice;
char t='y';
clrscr();
do
{
cout<<"\n 1.deposit 2.withdraw 3.display current balance"<<endl;
cin>>choice;
switch(choice)
{
case 1:
cout<<"enter deposit amount\n";
cin>>amt;
bank1.deposit(amt);
break;
case 2:
cout<<"enter withdraw amt:";
cin>>amt;
bank1.withdraw(amt);
break;
case 3:
bank1.display();
break;
}
cout<<"want to cont [y/n]";
cin>>t;
}while(t=='y');
}
```

Output :

```
1.deposit 2.withdraw 3.display current balance
1
enter deposit amount
500
want to cont [y/n]y

 1.deposit 2.withdraw 3.display current balance
3

current balance
 name:Rama
 balance600
want to cont [y/n]y
```

```
 1.deposit 2.withdraw 3.display current balance
2
enter withdraw amt:150
want to cont [y/n]y

 1.deposit 2.withdraw 3.display current balance
3

current balance
 name:Rama
 balance450
want to cont [y/n]n
```

Binary Operator Overloading

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string
{
char str[100];
public:
string()
{
strcpy(str,"  ");
}
void getdata();
void putdata();
string operator+(string b);
int operator==(string b);
int operator<(string b);
};
void string::getdata()
{
cout<<"enter a string"<<endl;
cin>>str;
}
void string::putdata()
{
cout<<"\n"<<str<<endl;
}
string string::operator+(string b)
{
string c;
strcat(str,b.str);
strcpy(c.str,str);
return c;
}
int string::operator==(string b)
```

```
{
if(strcmp(str,b.str)==0)
return (1);
else
return(0);
}
int string::operator<(string b)
{
if(strcmp(str,b.str)<0)
return (1);
else
return(0);    }
void main()
{
string s1,s2,s3;
char c='y';
clrscr();

do
{
cout<<"enter two strings"<<endl;
s1.getdata();
s2.getdata();
s1.putdata();
s2.putdata();
if(s1==s2)
{
cout<<"\nstring are equal";
goto l;
}
else
cout<<"\nnot equal";
s3=s1+s2;
s3.putdata();

if(s1<s2)
cout<<"\nstring1 are less than string2";
else
cout<<"\nstring2 is less than string1";
l:
cout<<"\ndo you want to cont [y/n]";
c=getch();
}
while(c=='y');

getch();
}
```

Output:

```
enter two strings
enter a string
ss
enter a string
ss

ss
ss
string are equal
do you want to cont [y/n]
enter two strings
enter a string
new
enter a string
delhi

new
delhi
not equal
newdelhi
string2 is less than string1
do you want to cont [y/n]
```

Virtual Functions

```
#include<iostream.h>
#include<conio.h>
class area
{
public:
float a;
virtual float area1(float dim)
{
a=dim;
return(a);
}
virtual void disp()
{
cout<<"dim is"<<a<<"\n";
}
};
class squ:public area
{
public:
float area1(float dim)
{
a=dim;
```

```
a=a*a;
return(a);
}
void disp()
{
cout<<"\narea of squre"<<a<<endl;
}
};
class circle:public area
{
public:
float area1(float dim)
{
a=dim;
a=a+a*2217;
return(a);
}
void disp()
{
cout<<"\narea of circle"<<a<<endl;
}
};

class cube:public area
{
public :
float area1(float dim)
{
a=dim;
a=a*a*a;
return(a);
}
void disp()
{
cout<<"\narea of cube"<<a<<endl;
}
};
void main(void)
{
clrscr();
float dim;
cout<<"enter dim";
cin>>dim;
area ar;
area*ptr;
ptr=&ar;
squ obj1;
circle obj2;
cube obj3;
ptr->area1(dim);
```

```
ptr->disp();
ptr=&obj1;
ptr->area1(dim);
ptr->disp();
ptr=&obj2;
ptr->area1(dim);
ptr->disp();
ptr=&obj3;
ptr->area1(dim);
ptr->disp();
getch();
}
```

Output:

```
enter dim5
dim is 5
area of squre25
area of circle11090
area of cube125
```

Implementation of Multiple Inheritance

```
#include<iostream.h>
#include<conio.h>
class empinfo
{
protected:
char empname[20];
int empno;
public:
void getempinfo();
void printempinfo();
};
void empinfo::getempinfo()
{
cout<<"enter employeename";
cin>>empname;
cout<<"enter employee number";
cin>>empno;
}
void empinfo::printempinfo()
{
cout<<"name is"<<empname;
cout<<"number is"<<empno;
}
class empearn
{
protected:
```

```
   int bp,hra,cca;
public:
  void getempearn();
  void printempearn();
  };
void empearn::getempearn()
{
cout<<"enter basic pay";
cin>>bp;
cout<<"enter hra";
cin>>hra;
cout<<"enter cca";
cin>>cca;
}
void empearn::printempearn()
{
cout<<"\nbasic pay is"<<bp;
cout<<"\nhra is"<<hra;
cout<<"\ncca is"<<cca;
}
class empdedn
{
protected:
    int lic,pf;
public:
    void getempdedn();
    void printempdedn();
    };
void empdedn::getempdedn()
{
cout<<"enter lic";
cin>>lic;
cout<<"enter pf";
cin>>pf;
}
void empdedn::printempdedn()
{
cout<<"\nlic"<<lic;
cout<<"\npf"<<pf;
}
class empsal:public empinfo,public empearn,public empdedn
{
protected:
int totsal;
public:
void calcsal();
};
void empsal::calcsal()
{
 totsal=bp+hra+cca-(lic+pf);
```

```
cout<<"\nnetsalary is"<<totsal;
}
void main()
{
empsal e;
clrscr();
e.getempinfo();
e.getempearn();
e.getempdedn();
e.printempinfo();
e.printempearn();
e.printempdedn();
e.calcsal();
getch();
}
```

Output:

```
enter employeename raja
enter employee number123
enter basic pay5000
enter hra1000
enter cca500
enter lic500
enter pf200
name israjanumber is123
basic pay is5000
hra is1000
cca is500
lic500
pf200
netsalary is5800
```

Implementation of Multilevel Inheritance

```
#include<iostream.h>
#include<conio.h>
class student
{
protected:
 int roll_number;
public:
 void get_number(int);
 void put_number(void);
};
void student::get_number(int a)
{
```

```
roll_number=a;
}
void student::put_number()
{
cout<<"\nRoll Number: "<<roll_number<<"\n";
}
class test : public student
{
protected:
  float sub1;
  float sub2;
  public:
  void get_marks(float,float);
  void put_marks(void);
  };
  void test::get_marks(float x,float y)
  {
   sub1=x;
   sub2=y;
   }
  void test::put_marks()
  {
  cout<<"Marks in Sub1= "<<sub1<<"\n";
  cout<<"Marks in Sub2= "<<sub2<<"\n";
  }
class result:public test
{
float total;
public:
  void display(void);
  };

  void result:: display(void)
  {
   total = sub1 + sub2;
   put_number();
   put_marks();
   cout<<"Total= " <<total<<"\n";
   }
int main()
{
result s;
clrscr();
s.get_number(111);
s.get_marks(75.0,60.5);
s.display();
getch();
return 0;
}
```

Output:

Roll Number: 111
Marks in Sub1= 75
Marks in Sub2= 60.5
Total= 135.5