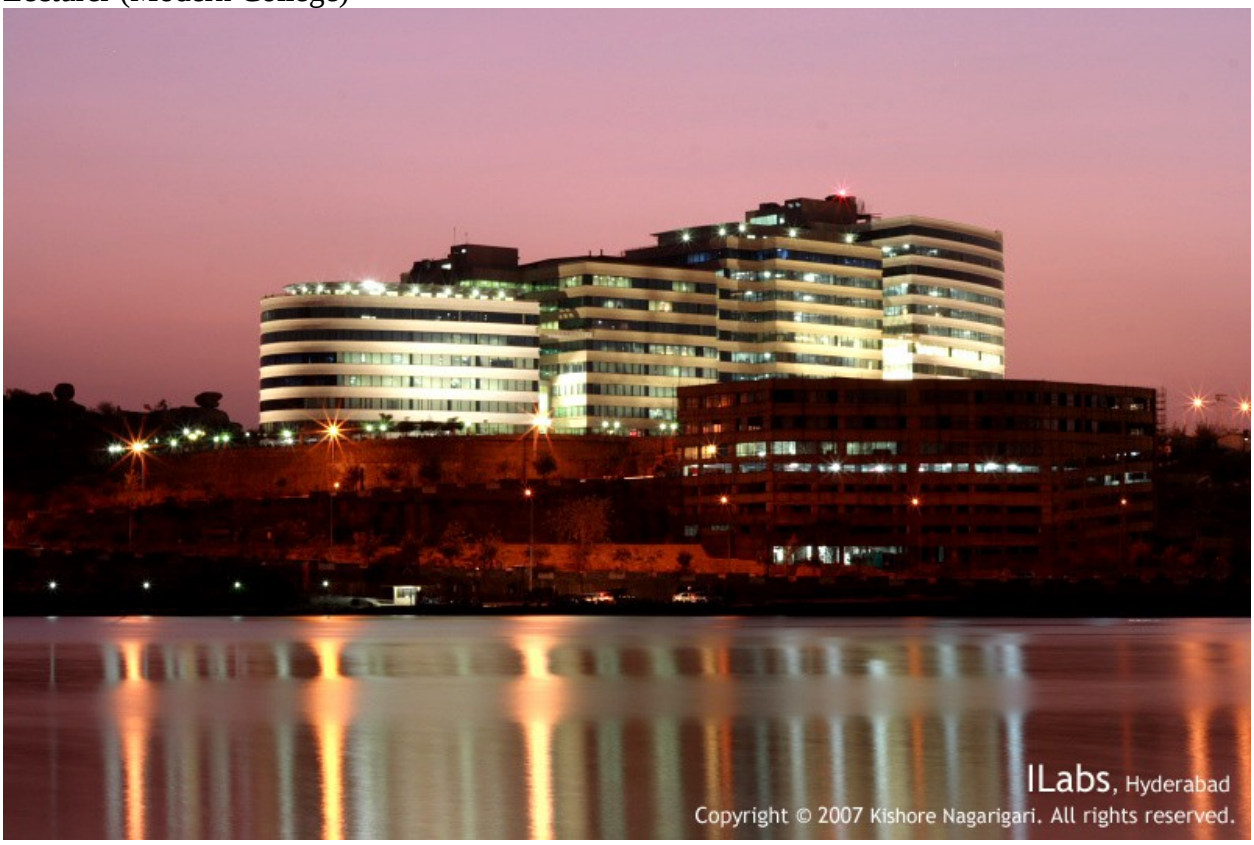
Rajesh S. Jha
Lecturer (Modern College)



Make your goal and achieve it. I assure you, I help you to make a path to achieve it. Let's begin.

Rajesh S. Jha
Lecturer (Modern College)

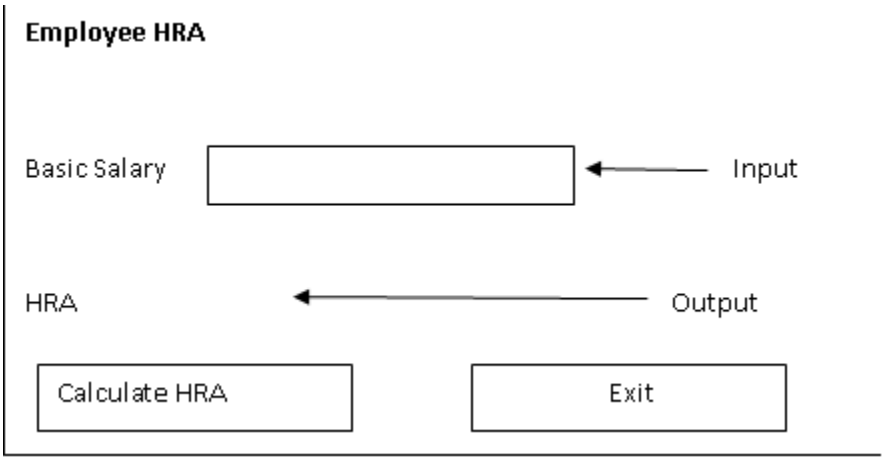
Programming Logic:

Problem Analysis/Problem Definition:

The first step in programming is to define the problem and understand it. Only then we can identified the exact problem and develop clear logic to solve that problem. This logic can then be used to create a computer application. Therefore, it is necessary to carefully assess the problem and identify the causes of the problem for us.
Problem definition requires identification of the following:

1. Tasks
2. Objects
3. Events

Following figure display the input and output object of HRA automation program.



Where: calculate HRA and Exit are Action Objects.

Process Analysis:

After defining the problem clearly, the user interface will created. The user interface is almost a replica of the sketch depicted the objects shown in above figure. While designing an interface we need to ensure that the interface should be easy to understand and well structured.
Case study:
The diagram of the HRA automation system describes the objects of the problem as shown in figure. The figure shows that a control is needed to accept the basic salary of the employees and one control to display the HRA. And two buttons is needed to calculate HRA and Exit from application environment.

Rajesh S. Jha
Lecturer (Modern College)

Conceptual development of solution:

Logical development for actions is one of the most important steps of programming approach. This step defines the events associated with each object and identifies the way our objects should respond to particular events.
Programming used two tools that help in logic development for the action objects:

- Input processing/output (IPO) tables and
- Pseudocode

Following table show input to an object, the output from an object, and the processing required for converting the input to the desired output. After creating IPO table, we can write pseudocode procedure.

A Pseudocode provides the link between an IPO table and the computer code for the action objects. Psedocode will replace the English statements with the equivalent instruction in a programming language.

INPUT	PROCESSING	OUTPUT
Basic Salary	HRA=17%*basic salary	HRA

Table: IPO Table

```
Void main ()
{
int basic, hra;
cin>>basic;
hra=0.17*basic;
cout<<basic;
}
```

After this step we first test our application under each and every condition. Debugging and testing our application. In the last create Documentation for our application.

Development Tools:

Algorithm:

Algorithm is a step by step instruction of a program. It is a finite set of step by step instructions that solve a problem. After defining the problem, we can create algorithms of our application.
Ex: short paper planning of any program.
Computer program is another example of an algorithm. Every computer program is a set of series in specific order, designed to perform a specific task.
Ex: Algorithm for largest no out of two no:

1. Start
2. Read no x and y from user
3. Check x>y if yes go to step 4 otherwise goto step 5
4. Print X is largest. Goto step 6

5. Print y is largest
6. Stop

Advantage:

1. An algorithm is generally written in spoken language and not in any programming language so hence it is easy to understand.
2. Also written an algorithm helps eliminate any error in our problem solving logic, so that when we start programming we worry only about the language specific issues.
3. Easy to understand
4. It support in writing program.

Disadvantage:

1. Algorithms are only as good as the instruction given. However, and the result will be incorrect if the algorithm is not properly defined.
2. Difficult to depict complex programming steps.
3. Most of the programmers write algorithm after writing the program; which fails the purpose of algorithm.

Flowcharts:

Flowchart is a pictorial representation of the steps involved in the procedure. It also shows the logical sequence, in which the steps are performed.
The flowchart is drawn by programmer to ensure the accuracy of their interpretation of the logic required in the program.
Flowchart is a common type of chart that represents an algorithm or process using different boxes and their arrows. Flowcharts are used in analyzing, designing, documentation or managing a process or program in various fields.

Advantage of flowchart:

1. Clarify the program logic.
2. Before coding begins, the flowcharts assist the programmer in determining the type of logic control to be used in a program.
3. Serve as documentation.
4. Serve as guide for program coding or program writing.
5. The flowchart is pictorial representation that may be useful to the business person or user who whishes to examine some aspects of the logic used in the program.

Disadvantage of flowchart:

1. Flowchart is a cumbersome for the programmer to write. As a result, many programmers do not write the chart until after the program has been completed, which defect one of the main process.
2. Flowchart is no longer completely standardized tool.
3. Some time it is not easily understood by common person and businessman.

Symbol	Function	Meaning

	Arrows	Direction of logic or flow
	Oval	Start/stop
	Rectangle	Processing/calculation
	Diamond shape	Decision
	Circle	connector
	Rhombus	Input/output

Pseudocode:

It is a tool used to formulate the processing steps of a program. Pseudocode uses English word to describe the processing steps of program or module.
The logic control instructions within a structured design are emphasized in pseudocode.
Ex: MOV is used for move, ADD is used for addition, and SUB is used for subtraction.

Advantage:

1. As compared to a flowchart, converting a pseudocode into a programming language is much easier.
2. As compared to a flowchart, it is easier to modify the pseudocode of program logic when program modifications are necessary.
3. Writing of pseudocode involves much less time and effort than drawing an equivalent flowchart.

Disadvantage:

1. In case of pseudocode, a graphic representation of program logic is not available.
2. There are no standard rules to follow in writing pseudocode. Different programmer uses their own style of writing pseudocode. Hence communication problem occur due to lack of standardization.
3. For beginner, it is difficult to follow the logic of pseudocode or write pseudocode as compared to flowcharting.

Programming structure:

Rajesh S. Jha
Lecturer (Modern College)
Programming structure is also called as control structure. And it is using only the following three simple logic (control) structures:

1. **Sequence logic**

2. **Selection logic**

3. **Iteration (or looping) logic**

Sequence logic:

In sequence logic, instructions are arrange in one after another in sequence. Thus in sequence logic, Pseudocode instruction are written in order, one after another from top to bottom.

(a) **Flowchart**

.
.
.
Process 1
.
.
.
Process 2

Selection Logic:

Selection logic also known as decision logic, used for making decision. It is used for selecting the proper path out of two or more alternative path in program logic.
Selection logic is classified into:

- If….then
- If…else
- Case structure

No

Yes

a) **Flowchart**
:
If condition
Then process 1
Else process 2
End if
:

Iteration (or Looping):

Iteration means repeated one statement again and again. Hence Iteration logic is used to produce loops in program logic when one or more instructions may be executed several times depending on some condition.
It Uses structure:

- DO….WHILE
- REPEAT….UNTIL
- FOR LOOP
- WHILE LOOP

no

yes

a) **flowchart**

:
Repeat
Process1
Process n
Until condition
:
:

Rajesh S. Jha
Lecturer (Modern College)

Language Evolution:

Before discussing about Language, it is necessary for us know about what is Computer? What is computer Programming?

Computer: it is an electronic machine which can solve mathematical and logical problem.
But computer can not solve problem itself. It must be programmed for finding the solution of that problem.

Program: A program is a set of instructions in some specific language to solve any problem.
Here specific computer means any computer language that computer can understand. There are so many computer languages available for computer programming such as basic, Pascal, COBOL, c, c++, java etc.

Types of programming languages:

1) **Low level language:**

Low level language is also called as Machine Language or binary language which is collection of very detailed instructions that control the computer internal circuits. It is machine dependent. Very few programs are written in this language because computers have their own instruction set. It is in the form of 0's and 1's.

A machine language instruction normally has a two parts format. The first part is operation code that tells the computer what function to perform, and the second part is operand that tells where to find or store the data to be manipulated.

OPCODE (operation code)	OPERAND (address/location)

(Instruction format)

Advantage:
Programs written in machine language can be executed very fast by a computer because machine instructions are understood by the computer without using any translator.

Limitation of machine language:

1) Machine dependent
2) Difficult to program
3) Error prone
4) Difficult to modify

2) **Assembly language**

Machine languages are very difficult for programming. However assembly languages uses mnemonic for each instruction in programs. Assembly language contains simple instruction like ADD A, SUB B etc.

Advantage:

- Easier to understand and used
- Easier to locate and correct errors
- Easier to modify
- No worry about addresses
- Easily reloadable
- Efficiency of machine language.

Disadvantage:

- Machine dependent
- Knowledge of hardware required
- Machine level coding

3) **High level language:**

Above two type of language are not easy for a programming. To make programming simpler high level languages are developed. These languages contain statements that are similar to a English language and easy to learn and remember.
Pascal, COBOL, FORTRAN etc are High level languages. These are machine dependent. There are three significant advantages in high level language:-

a) Simplicity
b) Uniformity
c) Profitability (machine independent)
Single instruction in high level language will be equivalent to many instructions in machine language. A high level language can generally be run on many different computers.

Advantage:

- They are machine independent.
- They do not required programmer to know anything about internal structure of the computer on which high level programs are executed.
- They do not deal with machine level coding.
- Easier to learn and use
- Fewer error
- Lower program preparation cost
- Better documentation
- Easier to maintain

Disadvantage:

1) Lower efficiency

2) Less flexibility

Translators:

As it is easier for programmer to learn any high level language and write the program using a HLL. Because it is much easier to write program and code it. As we know that computer can not understand any other language other than it own machine language (binary language) therefore, it is necessary to process a HLL program so as to make it understandable to the computer.
The system programs (software) that perform this very job are language processors.
These are as follows:

1) Assemblers
2) Interpreters
3) Compilers

The job of all these are same i.e. to convert the source code (any HLL program code) into the object code (MLL code) but the way is different.

1) **Assembler:**
   The translation program that translates an assembly code into the computer's machine code is called Assembler.
   Assembly language program (Source program) input output Machine language program (Object program)

2) **Interpreter**
   It translates a HLL program into machine language by converting and executing it line by line. If there is any error in any line, it reports it at the same time and program execution cannot resume until the error is rectified.
   Interpreter must always be present in the memory every time the program executed as every time the program is run, it is first interpreted and then executed. For error debugging, interpreter is very much useful as it reports the errors at the same time. But once the error are rectified unnecessary usage of memory takes pace as it always has to be present in the memory.
   Ex: BASIC language program are interpreted.

3) **Compiler**
   A compiler also translated an HLL program into machine language but in a different way. It translates the entire HLL program in one go, and reports all the error of the program along with line number.
   After all the errors are rectified, the program is recompiled, and after that the compiler is not needed in the memory as the object program is available.
   high level language program input output Machine language program

(Source program) (Object program)

Object Code (object program): translation of the source code of a program into machine code, which the computer can read and execute directly. Object code is the input to the linker.

Linker: A program that links separately compiled function together into one program. The o/p of the linker is an executable program. It combines the function in the standard C library with the code that we write.

Library: The file containing the standard functions that can be used by wer program. In c header files are included. For ex getch(), clrscr().

Hence difference in compiler and interpreter is that an interpreter read the source-code of wer program one line at time but a compiler read the entire program and converts it into object code. Object code to also called binary code and Machine code.

Generation of Language:

A large number of high level languages have been developed since the first ones developed at early 1950's. After that, more than 1000 high level languages have been developed but many of them are no longer in use.
But all the language is categorized into two types:

1. **Procedure oriented programming language**
2. **Object oriented programming language**

FORTRAN:
FORTRAN stands for FORmula TRANslation. It is one of the oldest high-level languages. It was designed to solve scientific and engineering problems and it is currently most popular language among scientists and engineers. It was originally developed by John Backus and his team at IBM in 1957.

COBOL:
COBAL stand for common Business oriented language. It was designed for business data processing applications; today it is mostly used in business oriented applications. Business data processing application deals with storing, retrieving, and processing corporate accounting information.
It was designed and developed by retired navy commodore and mathematician Grace Hopper in 1959.

BASIC:
BASIC stand for Beginners All-purpose Symbolic Instruction Code. It was developed in 1964 by John Kemeny and Thomas Kurtz in US.

Rajesh S. Jha
Lecturer (Modern College)

Pascal:

Pascal is a language that provide programmer to write well-structured, modular programs, and overall provide good programming feature compare to previous developed language.
It was developed by French mathematician Blaise Pascal in 1971.

C and C++:

C language was developed by Dennis Ritchi and Brian Karnighan in 1972 at AT & T's bell Laboratories.
C is a procedure oriented programming language.
C++ is a object oriented programming language developed by Bjarne Stroustrup at bell labs in early 1980's.

Characteristics of good programming language:

- Simplicity
- Naturalness
- Abstraction
- Efficiency
- Structured programming support
- Compactness
- Locality
- Extensibility
- Suitability to its environment

C++ has the following characteristics over the other languages:

- **Object-oriented programming**
- **Portability**
- **Brevity**
- **Modular programming**
- **C Compatibility**
- **Speed**

Simple Input/Output operation in c++:

The most common way in which a program communicates with the outside world is through simple, character-oriented Input/Output (IO) operations. C++ provides two useful operators for this purpose: >> for input and << for output. We have already seen examples of output using <<. Listing 1.4 also illustrates the use of >> for input.

```
#include <iostream.h>
int main (void)
{
int workDays = 5;
float workHours = 7.5;
float payRate, weeklyPay;
```

```
cout << "What is the hourly pay rate? ";
cin >> payRate;
weeklyPay = workDays * workHours *
payRate;
cout << "Weekly Pay = ";
cout << weeklyPay;
cout << '\n';
}
```

How C++ Compilation Works?

Compiling a C++ program involves a number of steps (most of which are transparent to the user):

- First, the C++ **preprocessor** goes over the program text and carries out the instructions specified by the preprocessor directives (e.g., #include). The result is a modified program text which no longer contains any directives.

- Then, the C++ **compiler** translates the program code. The compiler may be a true C++ compiler which generates native (assembly or machine) code, or just a translator which translates the code into C. In the latter case, the resulting C code is then passed through a C compiler to produce native object code. In either case, the outcome may be incomplete due to the program referring to library routines which are not defined as a part of the program. For example, figure refers to the << operator which is actually defined in a separate IO library.

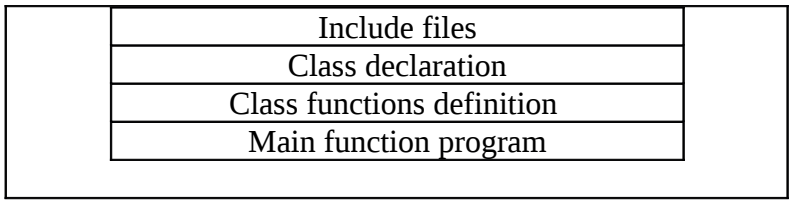
- Finally, the **linker** completes the object code by linking it with the object code of any library modules that the program may have referred to. The final result is an executable file.

  Figure illustrates the above steps for both a C++ translator and a C++ native compiler. In practice all these steps are usually invoked by a single command (e.g., CC) and the user will not even see the intermediate files generated.

Rajesh S. Jha
Lecturer (Modern College)

Explain the structure of general C++ program?

A typical c++ program contains four section as shown in following figure. These sections may be placed in different code files and then compiled independently or jointly.



Structure of c++ program

It is a common practice to organize a program into three separate files. The class declaring are placed in header file and definition of the member go in other file.
These approaches enable the programmer to separate the abstract of the interface from the implementation details. Finally the main program that uses the class is placed in third file, which include the previous two files as well as any other files required.
Ex:

```
class Point
{
       int xVal, yVal;
           public:
           void SetPt (int, int);
            void OffsetPt (int, int);
};
void Point::SetPt (int x, int y)
{
      xVal = x;
     yVal = y;
}
void Point::OffsetPt (int x, int y)
{
         xVal += x;
          yVal += y;
}
Void main()
{
Point pt; // pt is an object of class Point
pt.SetPt(10,20); // pt is set to (10,20)
pt.OffsetPt(2,2); // pt becomes (12,22)
getch();
}
```

End of 1st unit

Object oriented methodology:

As we know that C++ is a purely object oriented programming language. it is necessary to understand its foundational principles. OOP is a powerful way to approach the job of programming. The 1960s gave birth to structured programming. This is the method encouraged by languages such as C and Pascal.

Definition:

Object oriented programming (OOP) is an approach to program organization and development that attempt to eliminate some of the pitfalls of conventional programming method by using with the best structured programming features with several powerful new concepts.

Rajesh S. Jha
Lecturer (Modern College)
To support the principles of object-oriented programming, all OOP languages have three traits in common:

- Encapsulation,
- Polymorphism,
- Inheritance.

OOPs concepts are:

1) Object
2) Class
3) Encapsulation
4) Abstraction
5) Polymorphism
6) Inheritance
7) Message passing
8) Dynamic binding

Object oriented programming organizes a program around its data i.e objects and a set of well defined interface to that data. An Object-oriented program can be characterized as data controlling access to code.

BASIC CONCEPT/CHARACTERISTICS OF OOPS:

1. OBJECTS:
An object is an abstraction of a real world entity. It may represent a person, a place a number and icons or something else that can be modeled. Any data in an object occupy some space in memory and can communicate with each other.

2. CLASSES:
A class is a collection of objects having common features .It is a user defined data types which has data
Members as well functions that manipulate these data.

3. ABSTRACTION:
It can be defined as the separation of unnecessary details or explanation from system requirements so as to reduce the complexities of understanding requirements.

4. ENCAPTULATION:
It is a mechanism that puts the data and function together. It is the result of hiding implementation details of an object from its user .The object hides its data to be accessed by only those functions which are packed in the class of that object.

5. INHERITANCE:

It is the relationship between two classes of object such that one of the classes, the child takes all the relevant features of other class -the parent. Inheritance brings about reusability.

6. POLYMORPHISM:
Polymorphism means having many forms that in a single entity can takes more than one form. Polymorphism is implemented through operator overloading and function overloading.

7. DYNAMIC BINDING:
Dynamic binding is the process of resolving the function to be associated with the respective functions calls during their runtime rather than compile time.

8. MESSAGE PASSING:
Every data in an object in oops that is capable of processing request known as message .All object can
Communicate with each other by sending message to each other.

Advantage and application of Oops:

I. Emphasis is on data rather than procedure.
II. Programs are divided into number of objects.
III. Data structures are designed such that they characterize the objects.
IV. Function that operates on data of an object is tied together in the data structure.
V. Data are hidden and can not access by external function
VI. Objects may communicate with each other through function.
VII. New data and function can be easily added wherever required.

What is c++? What are advantages of c++?

C++ is a object oriented programming language. Initially C++ was named as "C with classes".

C was developed by Bjarne Stroustrup at AT & T bell library in USA, in early eighties.

The advantages of C++ over C:

- C++ is an incremented version of C. It is a superset of C. Almost All C programs can also run in c++ compiler.
- The important facilities added in C++ are classes, Function overloading, operator overloading.
- C++ allow user to create abstract data types, to inherit properties from existing data types.
- C++ supports polymorphism.
- Any real life application system such as editor, compiler, and databases communication systems can be built by C++.
- Object oriented library builts in C++.
- C++ programs can be easily implemented, maintained and expanded.

Rajesh S. Jha
Lecturer (Modern College)

Differences between procedure oriented programming approach and object oriented programming approach?

The Differences between procedure oriented programming approach and object oriented programming approach is:

Traditional procedure oriented programming approach	**object oriented programming approach**
In this approach, the problem is viewed as a sequence of things to be done.	In this approach, the problem is decomposed into a number of entities called objects and then builds data and function around these entities.
Emphasis is on doing things.	Emphasis is on the data rather than procedure
Larger program are divided into smaller programs known as functions.	Programs are divided into entities known as objects.
Data move openly around the system from function to function.	Data is hidden and can not be accessed by external functions
Employs top down approach in program design	Follows bottom-up approach in program design

C++ Character Set

C++ is *case sensitive* - that is, upper case letters are considered to be different from the corresponding lower case letters

Rajesh S. Jha
Lecturer (Modern College)

Basic C++ character set

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
_ $ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
<HT> <VT> <NL> <FF> <SPACE>

Escape Sequences

Code	Character	Description
\\	\	Backslash
\'	'	Single Quote
\"	"	Double Quote
\?	?	Question Mark
\0	<NUL>	Binary 0
\a	<BEL>	Bell (Audible alert)
\b	<BS>	Back Space
\f	<FF>	Form Feed
\n	<NL>	New Line
\r	<CR>	Carriage Return
\t	<HT>	Horizontal Tab
\v	<VT>	Vertical Tab

Notes:

- These escape sequences are used within character or string literal constants. They each represent a single character.

Numeric Escape Sequences

Width	Code	Where
8 bit	\nnn	n = octal digit
8 bit	\xnn	n = hexadecimal digit
8 bit	\Xnn	n = hexadecimal digit
16 bit	\unnnn	n = hexadecimal digit
32 bit	\Unnnnnnnn	n = hexadecimal digit

Notes:

- These escape sequences are used within character or string literal constants. They each represent a 1, 2, or 4 bytes, as indicated by the width.

Tokens: -

In a C++ source program, the basic element recognized by the compiler is the "token." A token is source-program text that the compiler does not break down into component elements.

- keyword
- identifier
- constant
- string-literal
- operator
- punctuator

The keywords, identifiers, constants, string literals, and operators described in this section are examples of tokens. Punctuation characters such as brackets (**[ ]**), braces (**{ }**), parentheses (**( )**), and commas (**,**) are also tokens.

Keywords:-

The keyword implement specified C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables of other user-defined program element.

reserved keywords:

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:
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Identifiers:-

In C++, we use identifiers to name user created entities which may be:

- Variable
- Function
- Type e.g. a class

Rule of identifier:

- A valid identifier is a sequence of one or more letters, digits or underscores 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 any case they can begin with a digit.

Home work:
Which of the following represent valid identifiers?

```
identifier
seven_11
_unique_
gross-income
gross$income
2by2
default
average_weight_of_a_large_pizza
variable
object.oriented
```

What are the different data types in C++?

Data types in C++ are shown in figure below:

Rajesh S. Jha
Lecturer (Modern College)

C++ allow user to create new abstract data types, which can behave like any built in data type. These are called user defined data types. This includes structure, union, class.
C++ provides three built in data types which are integral, void and floating. Integral include int and char. Floating include float and double.
C++ provides three derived in data types which are array, function and pointer.

Type	Byte	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
Float	4	-3.4e38 to 3.4e38
Double	8	-1.7e308 to 1.7e+308
long double	10	

Constants:

There are two ways to create constants in c++:

1. Using qualifier const.
2. Defining set of integer constant using enum.

A value declares by const keywords can not modified by a program.
For example:
const int size=10;

This declares size as a integer constants having value 10, another method of naming integer constants is:

enum
{

X,
Y,
Z
};
This defines x, y, and z as integer constants with values 0, 1, and 2 respectively.
We may write enum {x=100, y=23, z=234}

Variable:

In c++, all the variable must be declare before they are used in executable statements. They can be declaring any where in the program. Just before use of variable, we can declare a variable. This makes writing a program much easier. It also makes the program easier to understand.
For example:

```
float x;
float sum=0;
sum=sum+x;
float avg;
avg=sum/1;
```

the only disadvantage is that all the variables can not be seen at a glance.

Dynamic initialization of variable:

C++ permits the initialization of variable at runtime. This is called dynamic initialization.
For example:
We can write,

```
float area=3.14*rad*rad;
float avg=sum/I;
```

Observe that declaration and initialization of variable can be done simultaneously at a place where that variable is used for the first time.
Dynamic initialization is extensively used in object-oriented programming.

Reference variable:

A reference variable provides an alternative name (alias) for a previously defined variable. A reference variable is created as follows:

Data_type &reference_name=variable_name;

For example:

```
Flaot total=100;
Float &sum=total;
```

Here total and sum refer to same data object in memory. If we change total by total=total+10, it will cause change in value of both total and sum to 110;
Similarly sum=0; will change value of both variable to zero. A reference variable must be initialized at the time of declaration.

Rajesh S. Jha
Lecturer (Modern College)

Type Casting:

Converting an expression of a given type into another type is known as *type-casting*.
We have already seen some ways to type cast:

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type.
For example:

```
short a=2000;
int b;
b=a;
```

Here, the value of a has been promoted from short to int and we have not had to specify any type-casting
operator.

Explicit conversion

C++ is a strong-typed language. Many conversions, especially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a; // c-like cast notation
b = int (a); // functional notation
```

The functionality of these explicit conversion operators is enough for most needs with fundamental data types.

Different types of Operator:

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators.

1) **Assignment (=)**

The assignment operator assigns a value to a variable.
a = 5;

This statement assigns the integer value 5 to the variable a.
a = b;
This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue).

```
// 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;
}
o/p: a:4 b:7
```

2) 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 we 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.

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);

```
// compound assignment operators
#include <iostream>
using namespace std;
int main ()
{
int a, b=3;
a = b;
a+=2; // equivalent to a=a+2
cout << a;
return 0;
}
```

4) Increase and decrease (++, --)

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.

Example 1	Example 2
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.

5) 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.

6) 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).

7) 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 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>
using namespace std;
int main ()
{
int a,b,c;
a=2;
b=7;
c = (a>b) ? a : b;
cout << c;
return 0;
}
```

o/p:
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.

8) 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.

9) Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

Operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR

~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

10) Explicit type casting operator

Type casting operators allow us 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++.

11) 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.

12) Other operators

Later in these tutorials, we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming. Each one is treated in its respective section.

a) Precedence of operators

Rajesh S. Jha
Lecturer (Modern College)
When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:
a = 5 + 7 % 2
we may doubt if it really means:
a = 5 + (7 % 2) // with a result of 6, or
a = (5 + 7) % 2 // with a result of 0
The correct answer is the first of the two expressions, with a result of 6. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++.

All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses signs (and), as in this example:
a = 5 + 7 % 2;
Might be written either as:
a = 5 + (7 % 2);
Or
a = (5 + 7) % 2;
Depending on the operation that we want to perform.

Manipulators:

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw.**
The **endl** manipulators, when used in an output statement, causes linefeed to be inserted. It has the same effect as using the newline character "\n".
Ex:

```
……
……
cout<<"m="<<m<<endl
        <<"n="<<p<<endl
        <<"p="<<p<<endl;
……
```

……
This would cause three lines of o/p, one for each variable. If we assume the values of the variables as 25, 14, and 175 respectively, the o/p will appear as follows:
m=25
n=14
p=175
it is import ants to note that this form is the ideal o/p. here the number are right justified.
This form of o/p is possible only if we can specify a common field width for all the number and force them to be printed right justified. The setw manipulator does this job. It used as follows:
Sum=m+n+p;
Cout<<setw(5)<<sum<<endl;
The manipulator setw(5) specifies a field width 5 for printings the value of the variable sum, this value is right justified within the fields as shown below:

		2	1	4

Example:

```
#include<iostream.h> #include<iomanip.h>
Int main()
{ int basic=950,allowance=95, total=1045;
cout<<setw(10)<<"Basic="<<stew(10)<<basic<<endl
        <<setw(10)<<"allowance="<<stew(10)<<allowance<<endl
        << setw(10)<<"total="<<stew(10)<<total<<endl;
retunn 0;
}
```

o/p:

```
basic=          950
allowns=        95
total=          1045
```

Scope resolution operator:

The operator :: is called scope resolution operator. C++ is a block structure language i.e. a c++ program may contained one block with in another block.
When variable is declare in program, scope extends from the points of declaration till the end of the block in which it is defined. The same variable name can be used to have different meaning in different blocks.
Consider the following segments of program:
……..

```
……..
{
int x=10;
……
…….
        {
        int x=1;
        …….. block 2          block 1
        …….
        }
…….
……..
}
```

Here block 2 contain in block1. Note that the declaration of variable in inner block hides the declaration of same variables in an outer block.
Scope resolution operator is used to uncover a hidden variable.

It takes the form:

```
:: variable_name
```

e.g:

```
        ……
…….
{
int x=10;
…..
        {
        int x=1;
        cout<<"local x is:"<<x;
        cout<<"global x is:"<<::x;
        }
        …….
}
```

o/p:
local x is:1
global x is:10

Expressions:

Rajesh S. Jha
Lecturer (Modern College)
An expression is a combination of operators, constants, and variables arranged as per the rule of language. It may also include function calls which return value. An expression may consist of one or more operands, and zero or more operators to produce a value.

Expression may be of following seven types:

- Constants expression
- Integral expression
- Float expression
- Pointer expression
- Relational expression
- Logical expression
- Bitwise expression

Now we discuss each in details:

Constants expression:

- Constants expression consist only constant values.
  Ex:
  15
  10+5/2.0
  'x'

Integral expression:

- Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions.
  Ex:
  M
  M*n-5
  5+int(2.0)
  M*'x'
  Where m and n are integer variables.

float expression:

Float expressions are those which, after all conversions, produced floating point results.
Ex:
X+y
X*y/10
5+float(10)
10.75

Pointer expression:

Pointer expressions produced address values.
Ex:
&m
Ptr

Ptr+1
"xyz"
Where m is variable and ptr is a pointer.

Relational expressions:

Relation expressions give results of type **bool** which takes a value true or false.
X<=y
A+b==c+d
M+n>100

Logical expression:

Logical expression combines two or more relational expressions and produces **bool** type results.
Ex:
a>b && x==10
x==10||y==5

Bitwise expression:

Bitwise expression is used to manipulate data at bit level. These are usually used for testing or shifting bits.
Ex:
X<<3 //shift three bit position to left
y>>1 //shift one bit position to right

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 *compound statement* 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; }

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 (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.

Pictorial representation on the next page:

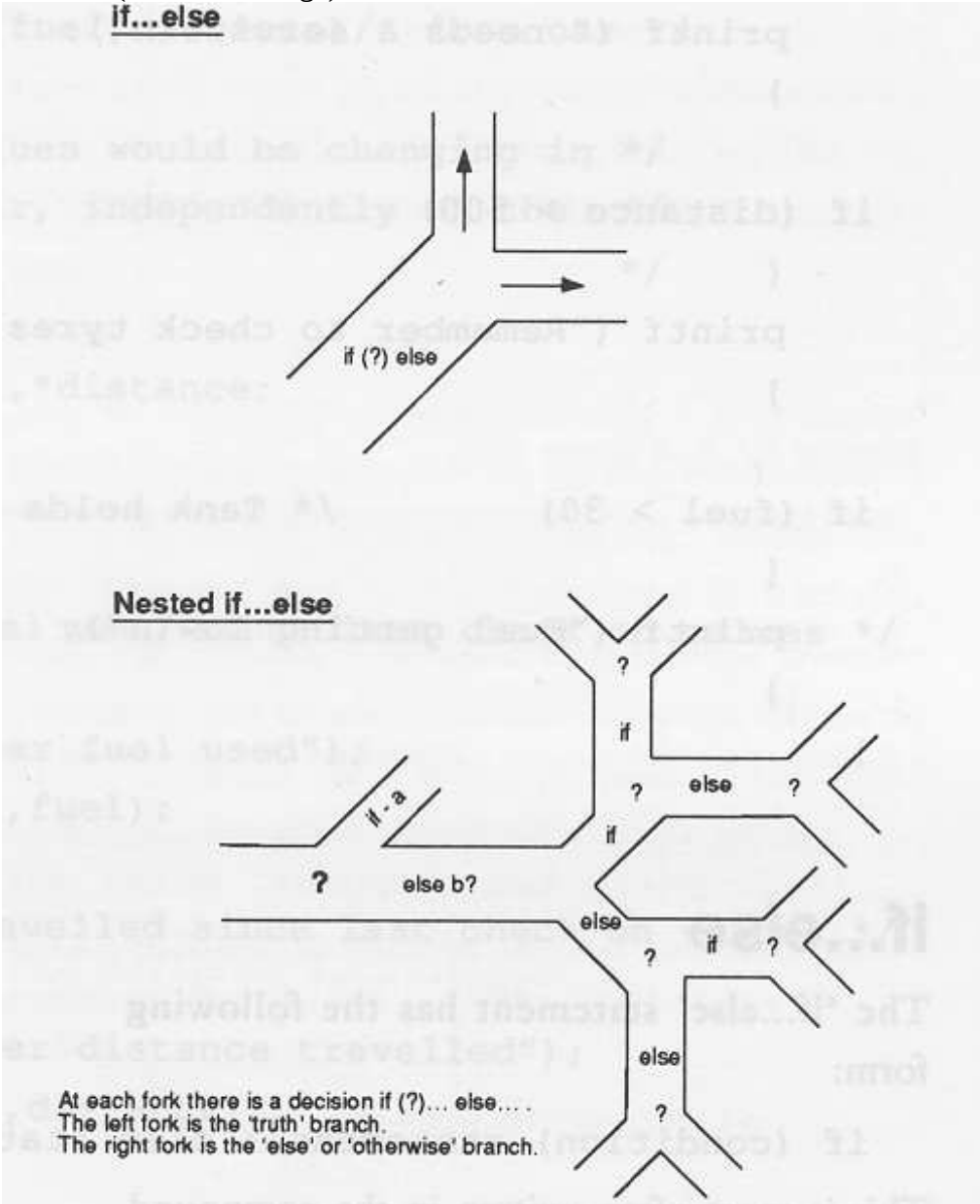


Figure 17.2. Which route – if...else selects.

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):

Rajesh S. Jha
Lecturer (Modern College)

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

or:

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

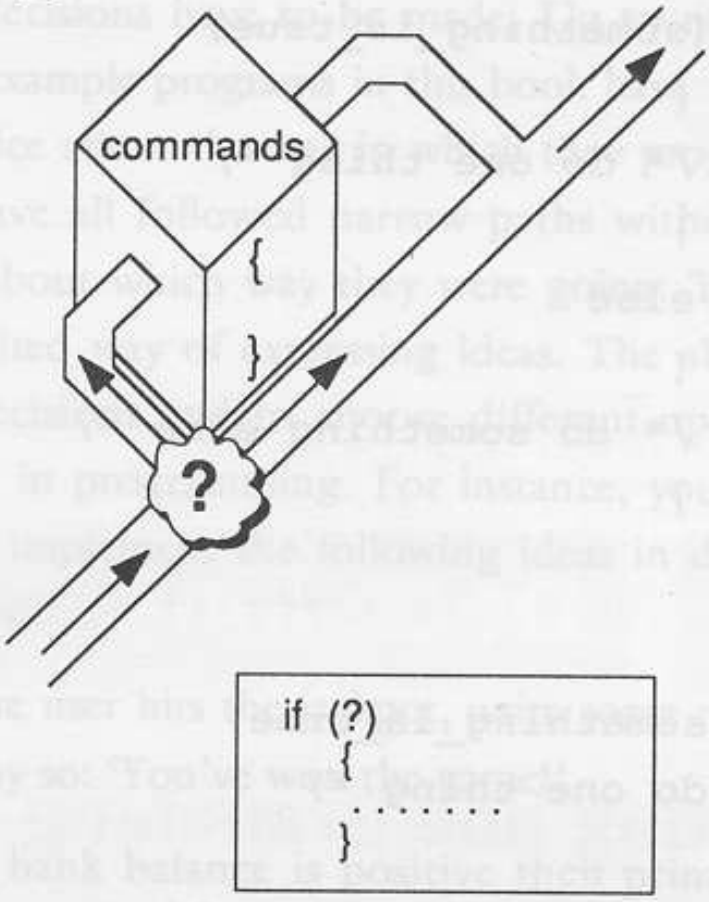


Figure 17.1. If some condition is satisfied, do the contents of this box then rejoin the main path.

```
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 { }.

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:

```
// custom countdown using while
#include <iostream>
using namespace std;
int main ()
{
int n;
cout << "Enter the starting number > ";
cin >> n;
while (n>0) {
cout << n << ", ";
--n;
}
cout << "FIRE!\n";
return 0;
}
```

o/p:
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 whole process of the previous program can be interpreted according to the following script (beginning in main):
1. User assigns a value to n
2. The while condition is checked (n>0). At this point there are two possibilities:
* Condition is true: statement is executed (to step 3)
* Condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:

cout << n << ", ";
--n;
(prints the value of n on the screen and decreases n by 1)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.

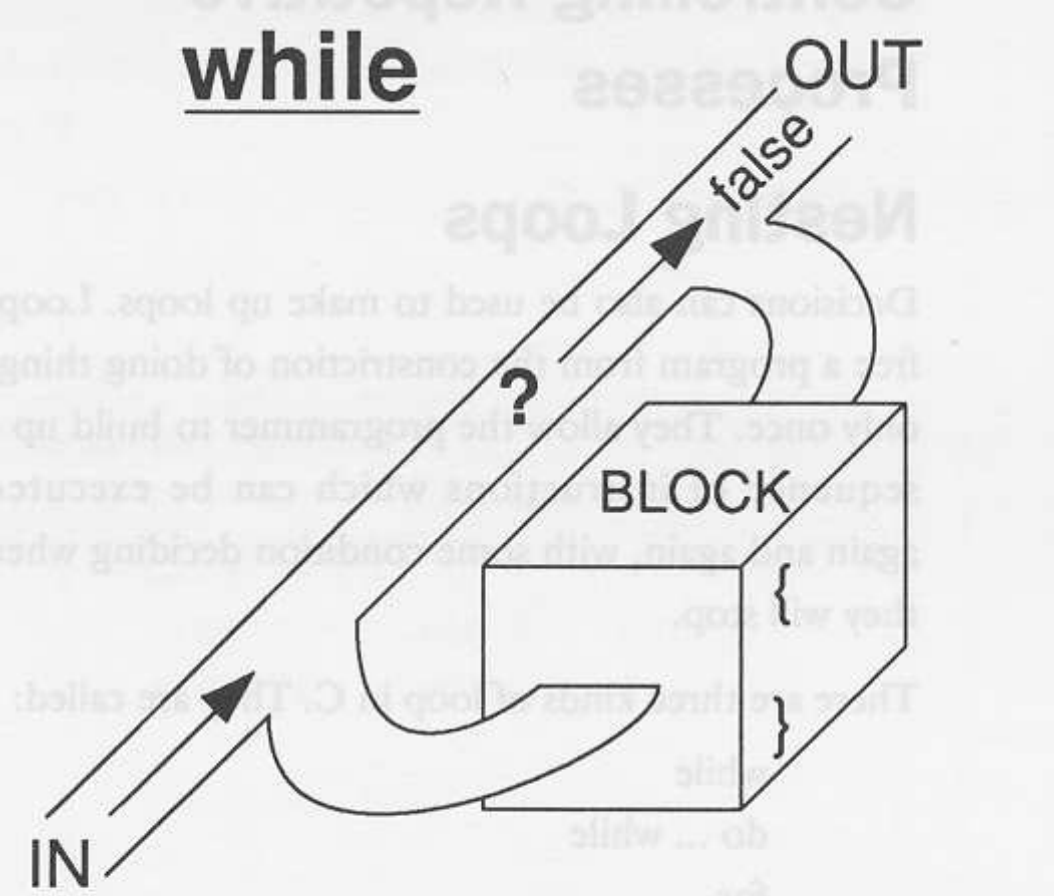


Figure 18.1. The structure of the while command.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever.
In this case we have included --n; that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) to become false after a certain number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end.
Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

Rajesh S. Jha
Lecturer (Modern College)

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 we enter until we enter 0.

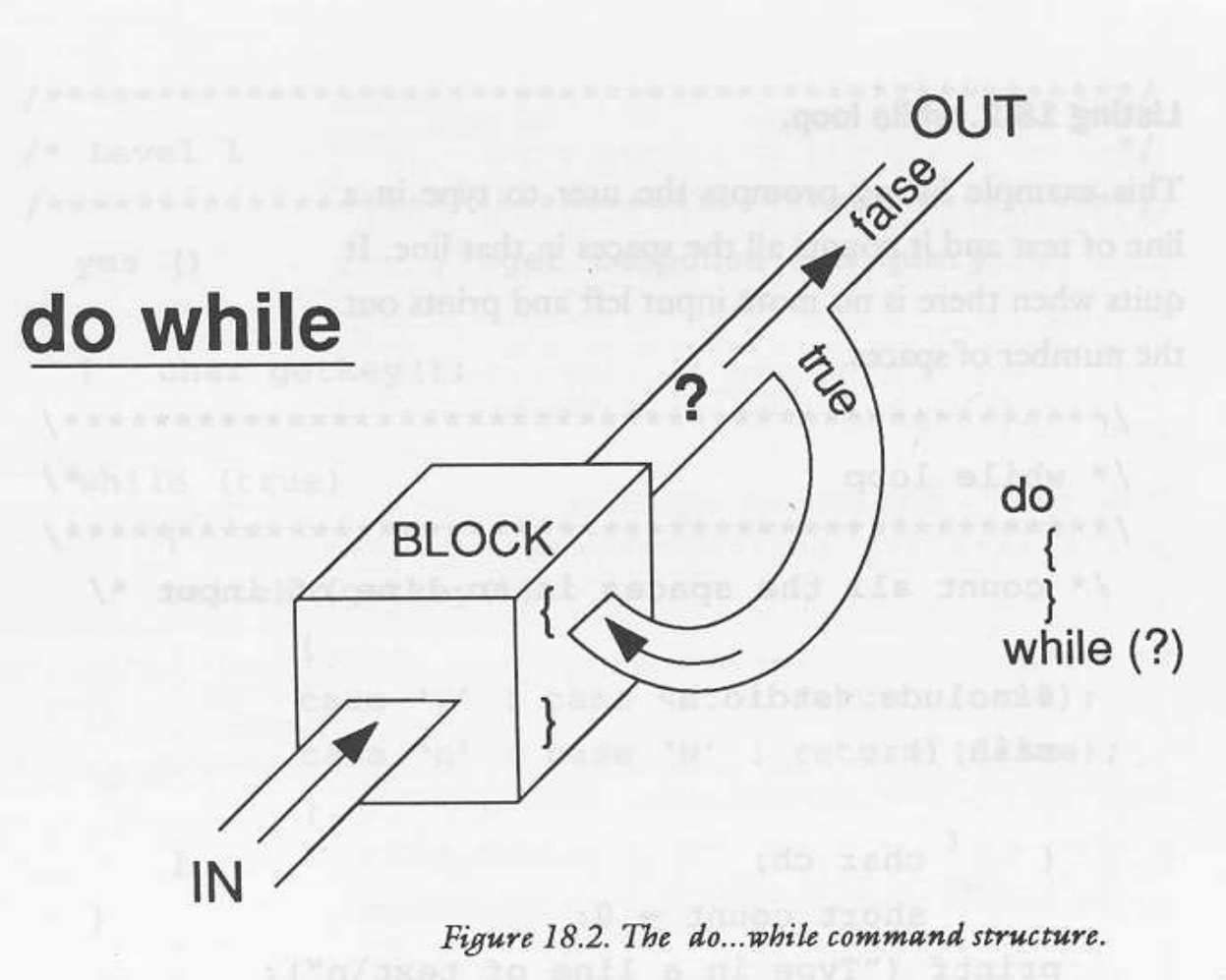


Figure 18.2. The do...while command structure.

```
// number echoer
#include <iostream>
using namespace std;
int main ()
{
```

```
unsigned long n;
do {
cout << "Enter number (0 to end): ";
cin >> n;
cout << "We entered: " << n << "\n";
} while (n != 0);
return 0;
}
```
```
Enter number (0 to end): 12345
We entered: 12345
Enter number (0 to end): 160277
We entered: 160277
Enter number (0 to end): 0
We entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the 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 we never enter the value 0 in the previous example we can be prompted for more numbers forever.

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.

Rajesh S. Jha
Lecturer (Modern College)

Here is an example of countdown using a for loop:

```
/* Calculation of simple interest for 3 sets of p, n and r */
main ( )
{
int p, n, count ;
float r, si ;
for ( count = 1 ; count <= 3 ; count = count + 1 )
{
Cout<< "Enter values of p, n, and r " ;
Cin>>p;
Cin>>n;
Cin>>r;
si = p * n * r / 100 ;
printf ( "Simple Interest = Rs.%f\n", si ) ;
}
}
```

2nd example:

```
// countdown using a for loop
#include <iostream>
using namespace std;
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 initialization, for example.

The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected.

For example, suppose that we wanted to initialize more than one variable in our loop:

```
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.

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>
using namespace std;
int main ()
{
int n;
for (n=10; n>0; n--)
{
cout << n << ", ";
if (n==3)
```

```
{
cout << "countdown aborted!";
break;
}
}
return 0;
}
```
o/p:
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>
using namespace std;
int main ()
{
for (int n=10; n>0; n--) {
if (n==5) continue;
cout << n << ", ";
}
cout << "FIRE!\n";
return 0;
}
```
o/p:
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

The goto statement:

goto allows to make an absolute jump to another point in the program. We 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 followed by a colon (:).
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:

```
// goto loop example
```

```
#include <iostream>
using namespace std;
int main ()
{
int n=10;
loop:
cout << n << ", ";
n--;
if (n>0) goto loop;
cout << "FIRE!\n";
return 0;
}
```

o/p:
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.

```
Ex:     main()
        {
        int I;
                for(i=0;i<=5;i++)
                { cout<<i;
                        If(i==3)
                        {
                        Cout<<"I am exit";
                        exit(1);
                        }
                }
        }
```

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;
.
.
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 (we can include as many case labels as values we 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";
}
```

Rajesh S. Jha
Lecturer (Modern College)

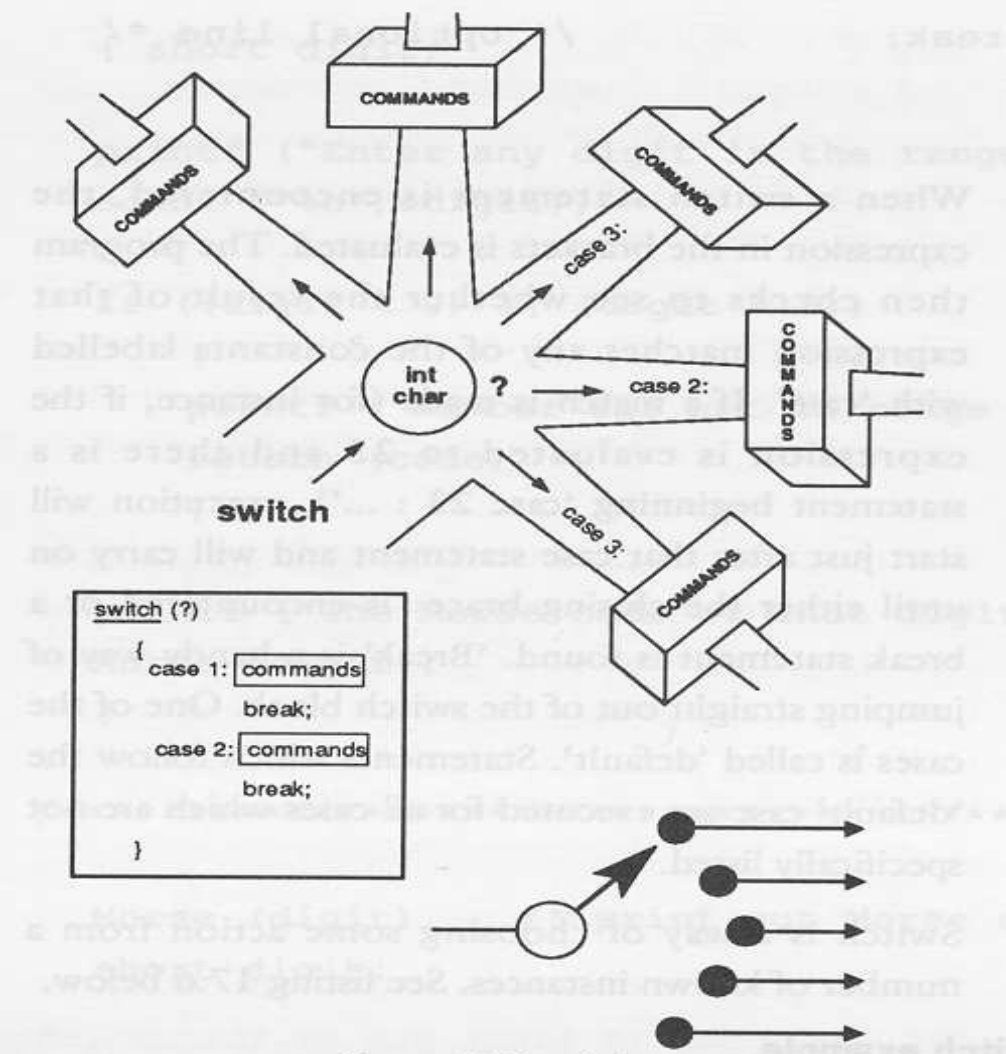


Figure 17.3. switch.

Now by using if statements:

```
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.

For example:

```
switch (x) {
case 1:
case 2:
case 3:
cout << "x is 1, 2 or 3";
break;
default:
cout << "x is not 1, 2 nor 3";
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example case n: where n is a variable) or ranges (case (1..3):) because they are not valid C++ constants. If we need to check ranges or values that are not constants, use a concatenation of if and else if statements.

Basic Input/Output:

Until now, the example programs of previous sections provided very little interaction with the user, if any at all. Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard.
C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen or the keyboard. A stream is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentially.
The standard C++ library includes the header file iostream, where the standard input and output stream objects are declared.

Standard Output (cout)

Rajesh S. Jha
Lecturer (Modern College)
By default, the standard output of a program is the screen, and the C++ stream object defined to access it is cout. cout is used in conjunction with the *insertion operator*, which is written as << (two "less than" signs).

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120; // prints number 120 on screen
cout << x; // prints the content of x on screen
```

The << operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout.
Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names.

For example, these two sentences have very different results:

```
cout << "Hello";       // prints Hello
cout << Hello;         // prints the content of Hello variable
```

The insertion operator (<<) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

This last statement would print the message Hello, I am a C++ statement on the screen. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my zipcode is " << zipcode;
```

If we assume the age variable to contain the value 24 and the zipcode variable to contain 90064 the output of the previous statement would be:

```
Hello, I am 24 years old and my zipcode is 90064
```

It is important to notice that cout does not add a line break after its output unless we explicitly indicate it,
therefore, the following statements:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them:
This is a sentence.This is another sentence. even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as \n (backslash, n):

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:
First sentence.
Second sentence.
Third sentence.

Additionally, to add a new-line, we may also use the endl manipulator. For example:
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
would print out:
First sentence.
Second sentence.

The endl manipulator produces a newline character, exactly as the insertion of '\n' does, but it also has an
additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an
unbuffered stream in most cases, so we can generally use both the \n escape character and the endl manipulator in order to specify a new line without any difference in its behavior.

Standard Input (cin).

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream.
For example:

```
int age;
cin >> age;
```

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable.cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if we request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced. We must always consider the type of the variable that we are using as a container with cin extractions. If we request an integer we will get an integer, if we request a character we will get a character and if we request a string of characters we will get a string of characters.

```
// i/o example
#include <iostream>
using namespace std;
```

```
int main ()
{
int i;
cout << "Please enter an integer value: ";
cin >> i;
cout << "The value we entered is " << i;
cout << " and its double is " << i*2 << ".\n";
return 0;
}
```
Please enter an integer value: 702
The value we entered is 702 and its double is 1404.

The user of a program may be one of the factors that generate errors even in the simplest programs that use cin (like the one we have just seen). Since if we request an integer value and the user introduces a name (which generally is a string of characters), the result may cause were program to misused since it is not what we were expecting from the user.

So when we use the data input provided by cin extractions we will have to trust that the user of were program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested.

A little ahead, when we see the string stream class we will see a possible solution for the errors that can be caused by this type of user input. We can also use cin to request more than one datum input from the user:

```
cin >> a >> b;
```
is equivalent to:
```
cin >> a;
cin >> b;
```
In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.

cin and strings

We can use cin to get strings with the extraction operator (>>) as we do with fundamental data type variables:
```
cin >> mystring;
```

However, as it has been said, cin extraction stops reading as soon as if finds any blank space character, so in this case we will be able to get just one word for each extraction. This behavior may or may not be what we want;
For example if we want to get a sentence from the user, this extraction operation would not be useful.

In order to get entire lines, we can use the function getline, which is the more recommendable way to get user input with cin:

```
// cin with strings
#include <iostream>
#include <string>
using namespace std;
int main ()
{
string mystr;
cout << "What's wer name? ";
getline (cin, mystr);
cout << "Hello " << mystr << ".\n";
cout << "What is wer favorite team? ";
getline (cin, mystr);
cout << "I like " << mystr << " too!\n";
return 0;
}
What's wer name? Juan SouliA¯A.A.
Hello Juan SouliA¯A.A..
What is wer favorite team? The Isotopes
I like The Isotopes too!
```

Notice how in both calls to getline we used the same string identifier (mystr). What the program does in the second call is simply to replace the previous content by the new one that is introduced.

Stringstream

The standard header file <sstream> defines a class called stringstream that allows a string-based object to be treated as a stream. This way we can perform extraction or insertion operations from/to strings, which is especially useful to convert strings to numerical values and vice versa.

For example, if we want to extract an integer from a string we can write:

```
string mystr ("1204");
int myint;
stringstream(mystr) >> myint;
```

This declares a string object with a value of "1204", and an int object. Then we use stringstream's constructor to construct an object of this type from the string object. Because we can use stringstream objects as if they were streams, we can extract an integer from it as we would have done on cin by applying the extractor operator (>>) on it followed by a variable of type int.
After this piece of code, the variable myint will contain the numerical value 1204.

```
// stringstreams
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main ()
{
string mystr;
float price=0;
int quantity=0;
cout << "Enter price: ";
getline (cin,mystr);
stringstream(mystr) >> price;
cout << "Enter quantity: ";
getline (cin,mystr);
stringstream(mystr) >> quantity;
cout << "Total price: " << price*quantity <<
endl;
return 0;
}
Enter price: 22.25
Enter quantity: 7
Total price: 155.75
```

In this example, we acquire numeric values from the standard input indirectly. Instead of extracting numeric values directly from the standard input, we get lines from the standard input (cin) into a string object (mystr), and then we extract the integer values from this string into a variable of type int (quantity).

Using this method, instead of direct extractions of integer values, we have more control over what happens with the input of numeric values from the user, since we are separating the process of obtaining input from the user (we now simply ask for lines) with the interpretation of that input.

Therefore, this method is usually preferred to get numerical values from the user in all programs that are intensive in user input.

Array:

An array is a collection of similar type of data (elements) that has a common name. An array can hold any type of data and these are stored in adjacent location into memory.
For example if we want to arrange the marks obtained by 100 students in ascending order. In this case, there are two options: (1): Either constructs 100 variables to store the marks or (2):

construct one variable capable of holding all the 100 values. So I think it is better to construct a single variable for holding all the values.
X={1,2,3,4,5,6,7,88,76,54,45,33,23,22};

Properties of array:

1. An array is a collection of similar type of elements.
2. These elements can be all integers or all characters or all floats.
3. The type of an array is type of its elements.
4. The location of an array is the location of its 1st element(base address)

Type of Array:

1) **One dimensional array:**
   It takes following form:
   Syntax: Type array name [size]
   Where type indicates the type of an array element such as int, char, float etc.
   Eg. char m[8], int a[5], so on.
   Ex: m=WELCOME
   char m[7]

W	E	L	C	O	M	E
m[0]	m[1]	m[2]	m[3]	m[4]	m[5]	m[6]

2) **Two dimensional array:**
   The general form is:
   Type array name [row size][column size];
   Here two brackets are used, which specifies the row size and column size respectively.
   Two dimensional array must have two variables, one for number of rows and another for number of columns.
   Ex:
   int stud[4][2]={{101,56}, {102, 85}, {103,45}, {104,75}};

3) **Multidimensional array:**
   The c language allows three or more dimensional array. The general form of a multidimensional array is:
   Type array name [s1][s2][s3]……[sn];
   Where, si, i=1, 2, 3, 4, ……. , n
   Ex:

```
int n[3][4][2];
{
        {
                {2, 4},
                {7, 8},
                {3, 4},
                {5, 6};
        },
        {
```

```
                    {7, 6},
                    {3, 4},
                    {5, 3},
                    {2, 3};
            },
            {
                    {8, 9},
                    {7, 2},
                    {3, 4},
                    {5, 1};

            }
    };
```

Here, three dimensional arrays can be considered of as an array of arrays of arrays. The outer array has three elements, each of which is a two dimensional array of four one dimensional arrays. Each of which contains two integers. These elements are arranged in memory as follows:

0^{th} 2-dimensional array								1^{st} 2-dimensional array								2^{nd} 2-dimensional array							
2	4	7	8	3	4	5	6	7	6	3	4	5	3	2	3	8	9	7	2	3	4	5	1
65478								65494								65510							

A Simple Program Using Array:

Ex: write a program to find average marks obtained by a class of 30 students in a test.

```
main( )
 {
 int avg, sum = 0 ;
int i ;
 int marks[30] ; /* array declaration */
          for ( i = 0 ; i <= 29 ; i++ )
           {
           Cout<<"enter marks:";
           Cin>>marks[i] ; /* store data in array */
          }
          for ( i = 0 ; i <= 29 ; i++ )
          sum = sum + marks[i] ; /* read data from an array*/
avg = sum / 30 ;
cout<<"average marks"<<avg;
 }
```

There is a lot of new material in this program, so let us take it apart slowly.

Entering Data into an Array :

Here is the section of code that places data into an array:

```
for ( i = 0 ; i <= 29 ; i++ )
 {
cout<<"enter marks:";
cin>>marks[i] ;}
```

The **for** loop causes the process of asking for and receiving a student's marks from the user to be repeated 30 times. The first time through the loop, **i** has a value 0, so the **cin>>**function will cause the value typed to be stored in the array element **marks[0],** the first element of the array. This process will be repeated until i become 29. This is last time through the loop, which is a good thing, because there is no array element like **marks[30]**.

In **cin**>>function, we have used the "address of" operator (&) on the element **marks[i]** of the array. we are passing the address of this particular array element to the **cin**>>function, rather than its value; which is what **cin**>>requires.

Reading Data from an Array:

The balance of the program reads the data back out of the array and uses it to calculate the average. The **for** loop is much the same, but now the body of the loop causes each student's marks to be added to a running total stored in a variable called **sum**. When all the marks have been added up, the result is divided by 30, the number of students, to get the average.

```
for ( i = 0 ; i <= 29 ; i++ )
sum = sum + marks[i] ;
avg = sum / 30 ;
cout<<"average marks"<<avg;
```

Revision:

- An array is a collection of similar elements.
- The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
- An array is also known as a subscripted variable.
- Before using an array its type and dimension must be declared.
- An array its elements are always stored in contiguous memory locations.

Array Initialization:

Array Initialization means, initialize an array at the time of declaring it.
Following are a few examples that demonstrate this.

```
int num[6] = { 2, 4, 12, 5, 45, 5 } ;
```

int n[ ] = { 2, 4, 12, 5, 45, 5 } ;
float press[ ] = { 12.3, 34.2 -23.4, -11.3 } ;

Note the following points carefully:
(a) Till the array elements are not given any specific values, they are supposed to contain garbage values.
(b) If the array is initialised where it is declared, mentioning the dimension of the array is optional as in the 2nd example above.

Rules for array initialization:

1) If the number of initialiser is less than the number of elements in the array, then the remaining elements are set to zero.
   Ex: int x[4]={1,2,3};
   i.e. intx[4]={1,2,3,0};
2) It give an error, if we specify more initialiser than the number of elements in the array.
3) If initialiser have been provided for an array. It is not necessary to specify the array length. In this case the length is derived from the initialiser.
   Ex: x[4]={1, 2, 3,4,5}

Input of One-dimensional array:

One dimensional array can be read by using simple for loop where the control variable of the loop is used to read array element.

```
Eg: int x[20];
int i;
For(i=1;i<=20;i++)
Cin>>x[i];
```

Output of one dimensional array:

Array elements can be displayed using simple for loop.

```
Eg:
int x[20];
for(i=1;i<=20;i++)
cout<<x[i];
```

Input of Two-dimensional array:

For reading the elements of two dimensional arrays, the input statement is placed within the pair of nested for loop. Each control variable controls one of the subscript of that array.

```
Eg: int a[5][6];
```

```
int I,j;
for(i=1;i<=5;i++)
{
        for(j=1;j<=6;j++)
        {
        Cin>>a[i][j])
        }
}
```

Output of Two dimensional arrays:

To print the two dimensional array elements, the output statement is placed within the pair of nested for loop.

```
Ex:  int a[5][6];
int I,j;
for(i=1;i<=5;i++)
{       for(j=1;j<=6;j++)
        {cout<< a[i][j];
        cout<<endl;
        }
}
```

Array Elements in Memory:

Consider the following array declaration:

int arr[8] ;

What happens in memory when we make this declaration? 16 bytes get immediately reserved in memory, 2 bytes each for the 8 integers (under Windows/Linux the array would occupy 32 bytes as each integer would occupy 4 bytes). And since the array is not being initialized, all eight values present in it would be garbage values. This so happens because the storage class of this array is assumed to be **auto**. If the storage class is declared to be **static** then all the array elements would have a default initial value as zero. Whatever be the initial values, all the array elements would always be present in contiguous memory locations. This arrangement of array elements in memory is shown in following Figure.

elements	***11***	***23***	***32***	***12***	***23***	***43***	***56***	***55***
Memory address	***65508***	***65510***	***65512***	***65514***	***65516***	***65518***	***65520***	***65522***

End of 2nd unit

Rajesh S. Jha
Lecturer (Modern College)

Function:

A function is a subprogram that can act on data and return a value. Every C++ program has at least one function, i.e. main(). When our 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 jump to the body of that function.

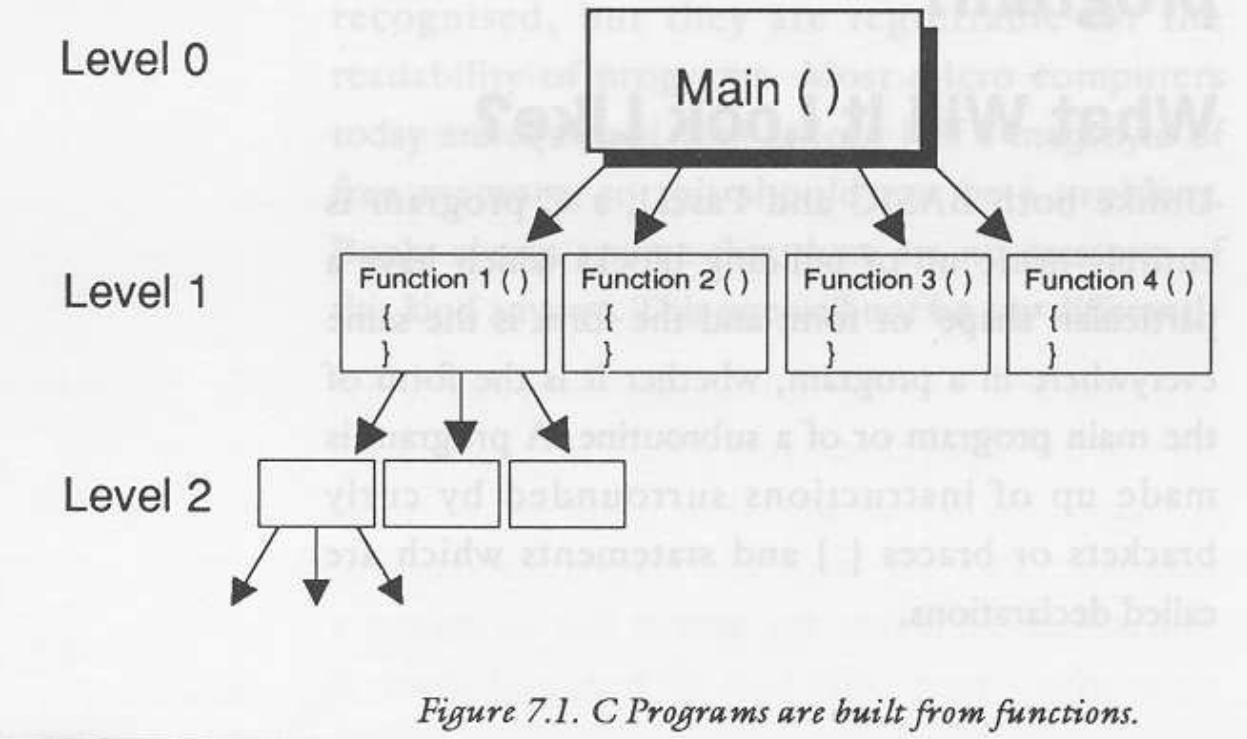


Figure 7.1. C Programs are built from functions.

The General Form of a Function:

The general form of a function is

Where,
The ***ret-type*** **specifies** the type of data that the function returns. A function may return any type of data except an array.

The ***parameter list*** is a comma-separated list of variable names and their associated types that receive the values of the arguments when the function is called. A function may be without parameters; in this case the parameter list is empty. However, even if there are no parameters, the

parentheses are still required. In variable declarations, we can declare many variables to be of a common type by using a comma-separated list of variable names.

In short, all function parameters must be declared individually, each including both the type and name.
That is, the parameter declaration list for a function takes this general form:

Function_name(*type varname1, type varname2, . . . , type varnameN)*

ex:
```
f(int i, int k, int j) /* correct */
f(int i, k, float j) /* incorrect */
```

ex:
```
void Show();   //function declaration
main()
{
-----------
Show();                  //function call
----------
Return 0;
}
Void show()    //function defination
{
//function body
}
```

User Defined function:

The function which is defined by user/developer i.e. which is not predefined is called user defined function.
Ex:
```
Main()
{
Cout<<"hello";
P_name();       //this is user defined function
Exit(0);                  // this is predefined function
Getch();         // this is predefined function
}
Void P_name()
{
Cout<<"my name is jha";
}
```

Function Prototyping:

The function prototype is a statement, which means it ends with a semicolon. It consists of the function's return type, name, and parameter list.
The parameter list is a list of all the parameters and their types, separated by commas.
The function prototype and the function definition must agree exactly about the return type, the name, and the parameter list. If they do not agree, we will get a compile-time error.

Syntax:
return_type function_ name(argument_list);
Argument_list contains the type and names of arguments that must be passed to the function.

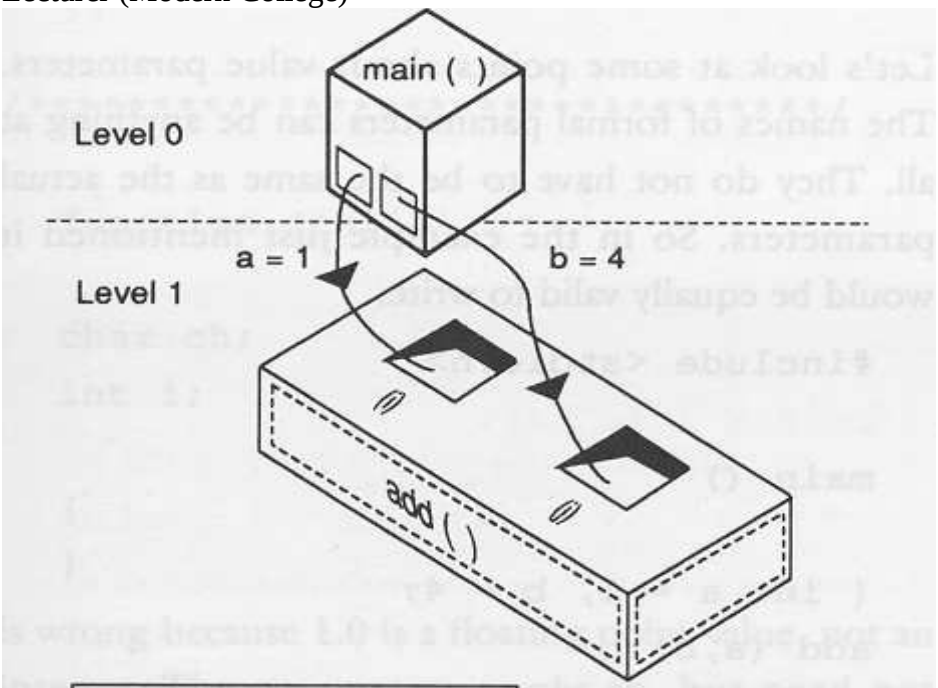
A prototype that looks like this is perfectly legal:
long Area(int, int);
This prototype declares a function named Area() that returns a long and that has two parameters, both integers. Although this is legal, it is not a good idea. Adding parameter names makes our prototype clearer.
The same function with named parameters might be:
long Area(int length, int width);

ex:

```
float volume(int x, float y, float z);
void main()
{
Float cube1=volume (10, 12.34, 50);
Cout<<cube1;
}
float volume(int a, float b, float c)
{
Float v=a*b*c;
}
```

Diagrammatic view:

Rajesh S. Jha
Lecturer (Modern College)




```
main ( )
{ int a = 1, b = 4
:
add (a,b);
}
```

```
add (i,j)
int i,j:
{
:
}
```

The variables i and j inside ( ) get the values from a and b but they can't send data back and change a or b.

Figure 11.1. Value parameters in action.

Function Prototype Examples

```
long FindArea(long length, long width);          // returns long, has two parameters
void PrintMessage(int messageNumber);    // returns void, has one parameter
int GetChoice();                         // returns int, has no parameters
BadFunction();                                   // returns int, has no parameters
```

Rajesh S. Jha
Lecturer (Modern College)

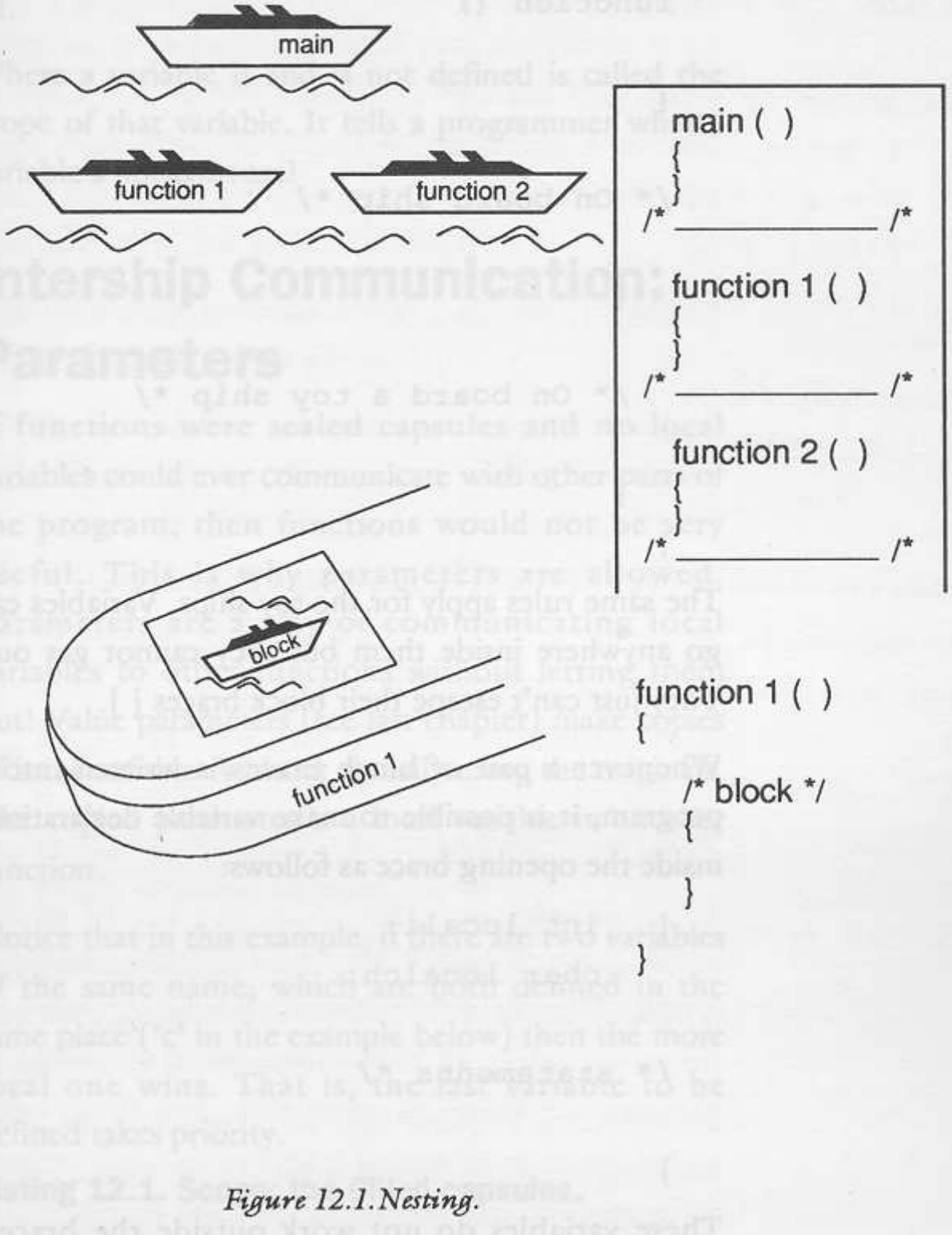


Figure 12.1. Nesting.

Function Definition Examples

```
long Area(long l, long w)
{
return l * w;
}
void PrintMessage(int whichMsg)
{
if (whichMsg == 0)
cout << "Hello.\n";
if (whichMsg == 1)
cout << "Goodbye.\n";
if (whichMsg > 1)
cout << "I'm confused.\n";
}
```

Call by reference and call by value:

A Function can be called by two methods:

1) Call by value
2) Call by reference

Arguments can be passed to a function either by **call by value and call by reference**.

Call by value:

- When function call passes arguments by value (call by value) the call function creates a new set of variables and copies the values of arguments into them.
- The function does not have access to the actual variables in calling program and can only work on the copies of values.
- Program for call by value:

```
#include <iostream.h>
int sqr(int x);
int main()
{
int t=10;
cout<<"square root is"<<sqr(t);        //call by value
return 0;
}
int sqr(int x)
{
x = x*x;
return(x);
```

}
In this example, the value of the argument to **sqr()** , 10, is copied into the parameter **x**. When the assignment **x = x*x** takes place, only the local variable **x** is modified. The variable **t**, used to call **sqr()** , still has the value 10. Hence, the output is **100 10**.
Remember that it is a copy of the value of the argument that is passed into the function. What occurs inside the function has no effect on the variable used in the call.

Call by reference:

- When function call passes arguments by reference (call by reference) the formal arguments in the called function become alises to the actual arguments in calling function. This means that when the function is working with its own arguments, it is actually working on the original data.

- The mechanism of call by value is good, if the function does not need to alter the value of the original variables in the calling program.

- But, if the situation to change the value of variables in calling program. I.e. in bubble sort compare two adjacent elements in the list and interchange them if first is greater than the second. In such situation, the function should be able to interchange the value of variable in calling programs, which is not possible by call by value. But it can be done if call by reference method is used.

- Program for call by reference:

```
void swap(int *x, int *y);
int main()
{
int i, j;
i = 10;
j = 20;
swap(&i, &j);  /* pass the addresses of i and j */
return 0;
}
void swap(int *x, int *y)
{
int temp;
temp = *x;       /* save the value at address x */
*x = *y;         /* put y into x */
*y = temp;       /* put x into y */
}
```

In this example, the variable **i** is assigned the value 10 and **j** is assigned the value 20. Then **swap()** is called with the addresses of **i** and **j**. (The unary operator **&** is used to produce the address of the variables.) Therefore, the addresses of **i** and **j**, not their values, are passed into the function **swap()** .

Returning by reference:

Function can also return a reference. It allows the function to be used on the left side of assingement statements.
For example consider the program:

```
char &replace(int i)            //return a reference
char s[80]="hello brother";
Main()
{replace(6)='x';
cout<<s;
}
char &replace(int i)
{
return s[i];
}
```

The output of above program is hello X brother. It replace hello and brother by X. in above program, replace (6) return 6^{th} elements i.e. a blank space, which is assigned to X.

Inline function:

Real concept behind inline function is:

When we define a function, normally the compiler creates just one set of instructions in memory. When we call the function, execution of the program jumps to those instructions, and when the function returns, execution jumps back to the next line in the calling function. If we call the function 10 times, our program jumps to the same set of instructions each time. This means there is only one copy of the function, not 10.

When programmers speak of efficiency, they usually mean speed: the program runs faster if the function call can be avoided.

If a function is declared with the keyword **inline**, the compiler does not create a real function: it copies the code from the inline function directly into the calling function. No jump is made; it is just as if we had written the statements of the function right into the calling function.

Definition:
Inline function is a function that is expanded in line when it is avoided i.e compiler replaces the function call with corresponding function code. The inline function has the form:

```
inline function_header
{
Function body;
}
```

Program:

```
#include <iostream.h>
inline float mul(float x, float y)
{
return(x*y);
}
inline double div(double p, double q)
{
return(p/q);
}
int main()
{
float a=12.345;
float b=9.82;
cout<<mul(a,b)<<"\n";
cout<<div(a,b)<<"\n";
return 0;
}
```

Output:
121.228
1.25713

Default and const arguments:

Default function arguments:

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function.
The default value is specified in a manner syntactically similar to a variable initialization.
For example, this declares **myfunc()** as taking one **double** argument with a default value of 0.0:

```
void myfunc(double d = 0.0)
{
// ...
}
```

Now, **myfunc()** can be called one of two ways, as the following examples show:

```
myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

The first call passes the value 198.234 to **d**. The second call automatically gives **d** the default value zero.
In another way we can say that C++ allow us to call a function without specifying all of its arguments. In such cases, the function assigns a default value to the parameter which does not have matching arguments in the function call. Default values are specified when function is declared.
For example:

```
float amount(float principal, int period, float rate=0.15);
```

above prototype declare on default value 0.15 to arguments rate.
A function call:

```
Value=amount(5000,7);
```

Passes the value 5000 to principal, 7 to periods and then lets the function use the default value 0.15 for rate.
The call:

```
Value=amount(5000,7,0.12);
```

Now Passes the value 5000 to principal, 7 to periods and then lets the function use the explicit value 0.15 for rate.
Note that trailing arguments can have default values i.e we must add defaults from right to left. We can not provided a default value to a arguments in the middle of list.

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 const tell the compiler that the function should not modify the arguments. The compiler will generates an error when condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

Function overloading:

Rajesh S. Jha
Lecturer (Modern College)

What is function overloading?
Function overloading is the ability to write more than one function with the same name, distinguished by the number or type of the parameters.

C++ enables we to create more than one function with the same name. This is called function overloading. The functions must differ in their parameter list, with a different type of parameter, a different number of parameters, or both.

Here's an example:
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);

- myFunction() is overloaded with three different parameter lists. The first and second versions differ in the types of the parameters, and the third differs in the number of parameters.
- The return types can be the same or different on overloaded functions. We should note that two
  functions with the same name and parameter list, but different return types, generate a compiler error.

Function *overloading* is also called function *polymorphism*. Poly means many, and morph means form: a polymorphic function is many-formed.

Ex:
//function volume () is overloaded three times.

1) #include<iostream.h>

```
//declaring prototypes
int volume(int);
double volume(double, int)
long volume(long,int,int);
int main()
{
cout<<volume(10)<<"\n";
cout<<volume(2,5,8)<<"\n";
cout<<volume(100L,75,8)<<"\n";
return 0;
}
```

```
//function definations
int volume(int s)        //cube
{
return(s*s*s);
}
double volume(double r, int h)                //cylinder
{
return(3.14*r*r*h);
}
long volume(long l, int b, int h)             //rectangular box
{
return(l*b*h);
}
```
Output:
1000
157.26
112500

2nd example of overloading:

```
#include <iostream>
using namespace std;
char myfunc(unsigned char ch);
char myfunc(char ch);
int main()
{
cout << myfunc('c'); // this calls myfunc(char)
cout << myfunc(88) << " "; // ambiguous
return 0;
}
char myfunc(unsigned char ch)
{return ch-1;}
char myfunc(char ch)
{
return ch+1;
}
```

In C++, **unsigned char** and **char** are *not* inherently ambiguous. However, when **myfunc()** is called by using the integer 88, the compiler does not know which function to call. That is, should 88 be converted into a **char** or an **unsigned char?**

3rd example:

```
#include <iostream>
using namespace std;
// abs is overloaded three ways
int abs(int i);
double abs(double d);
long abs(long l);
```

```
int main()
{
cout << abs(-10) << "\n";
cout << abs(-11.0) << "\n";
cout << abs(-9L) << "\n";
return 0;
}
int abs(int i)
{
cout << "Using integer abs()\n";
return i<0 ? -i : i;
}
double abs(double d)
{
cout << "Using double abs()\n";
return d<0.0 ? -d : d;
}
long abs(long l)
{
cout << "Using long abs()\n";
return l<0 ? -l : l;
}
```

The output from this program is shown here.

```
Using integer abs()
10
Using double abs()
11
Using long abs()
9
```

This program creates three similar but different functions called **abs(),** each of which returns the absolute value of its argument. The compiler knows which function to call in each situation because of the type of the argument.

Recursion:

A function can call itself. This is called recursion, and recursion can be direct or indirect. It is direct when a function calls itself; it is indirect recursion when a function calls another function that then calls the first function.

Ex:

a) To illustrate solving a problem using recursion, consider the Fibonacci series: 1,1,2,3,5,8,13,21,34…..

For example, recursion can be used to calculate the factorial of a number. The factorial of a number x is
written x! and is calculated as follows:
x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1
However, we can also calculate x! like this:
x! = x * (x-1)!
Going one step further, we can calculate (x-1)! using the same procedure:
(x-1)! = (x-1) * (x-2)!
We can continue calculating recursively until we're down to a value of 1.

Using a recursive function to calculate factorials:

```
#include <stdio.h>
unsigned int f, x;
unsigned int factorial(unsigned int a);
main()
{
puts("Enter an integer value between 1 and 8: ");
scanf("%d", &x);
if( x > 8 || x < 1)
{
printf("Only values from 1 to 8 are acceptable!");
}
else
{
f = factorial(x);
printf("%u factorial equals %u\n", x, f);
}
return 0;
}
unsigned int factorial(unsigned int a)
{
if (a == 1)
return 1;
else
{
a *= factorial(a-1);
return a;
}
}
```

Enter an integer value between 1 and 8:
6
6 factorial equals 720

Rajesh S. Jha
Lecturer (Modern College)

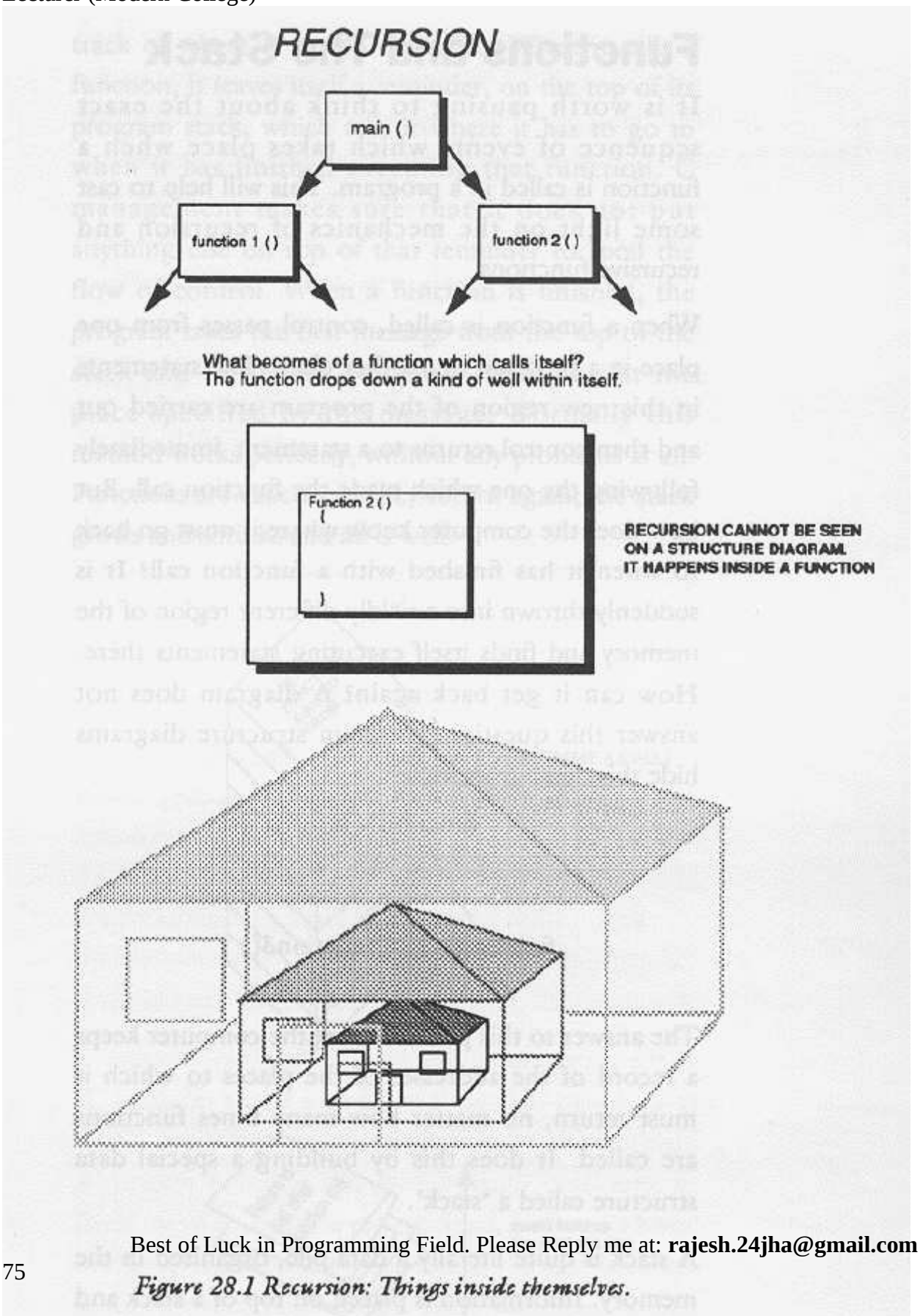


Figure 28.1 Recursion: Things inside themselves.

Rajesh S. Jha
Lecturer (Modern College)

Structure, Union and Enum:

Structure and union:

A structure contains a number of data types grouped together. These data types may or may not be of the same type. The following example illustrates the use of this data type.

```
main( )
{
        struct book
        {
        char name ;
        float price ;
        int pages ;
        } ;
struct book b1, b2, b3 ;
cout<< "\nEnter names, prices & no. of pages of 3 books\n" ;
cin>> b1.name>>b1.price>>b1.pages ;
cin>> b2.name>>b2.price>>b2.pages ;
cin>>b3.name>>b3.price>>b3.pages ;
cout<< "\n And this is what we entered"  ;
cout<< b1.name<< b1.price<< b1.pages ;
cout<< b2.name<<b2.price<< b2.pages ;
cout<< b3.name<<b3.price<< b3.pages ;
}
```

Output:

```
Enter names, prices and no. of pages of 3 books
A 100.00 354
C 256.50 682
F 233.70 512
And this is what we entered
A 100.000000 354
C 256.500000 682
F 233.700000 512
```

Definition : A structure is a heterogeneous user defined data type. it is a collection of logically related data items grouped together under a single name called as structure tag. The data items that makes up a structure are called its members or elements or fields or they can be different types.

Syntax:

```
struct tag
{
Member 1;
----------
Member n;
}
```

The above program demonstrates two fundamental aspects of structures:
(a) declaration of a structure
(b) accessing of structure elements

Rajesh S. Jha
Lecturer (Modern College)

Declaring a Structure:

The general form of a structure declaration statement is given below:

```
struct <structure name>
{
structure element 1 ;
structure element 2 ;
structure element 3 ;
......
......
} ;
```

In above example program, the following statement declares the structure type:

```
struct book
{
char name ;
float price ;
int pages ;
} ;
```

This statement defines a new data type called **struct book**. Each variable of this data type will consist of a character variable called **name**, a float variable called **price** and an integer variable called **pages**.
Once the new structure data type has been defined one or more variables can be declared to be of that type. For example the variables **b1**, **b2**, **b3** can be declared to be of the type **struct book**, as,

```
struct book b1, b2, b3 ;
```

This required 7 bytes—one for **name**, four for **price** and two for **pages**.

If we want, we can combine the declaration of the structure type and the structure variables in one statement.

For example,

```
struct book
{
char name ;
float price ;
int pages ;
} ;
struct book b1, b2, b3 ;
```

is same as...

```
struct book
{
char name ;
float price ;
int pages ;
} b1, b2, b3 ;
```

Structure initialization:

Like ordinary variable and array, structure variable can also initialized where they are declared:

```
void main()
{
struct student;
{
int roll;
int mark;
float per;
}
struct student std1={101,655,89.90};
cout<<“roll=”<<roll;
cout<<“mark of student=”<<mark;
cout<<“percentage=”<<per;
}
```

Or

```
void main()
{
struct student;
{
int roll;
int mark;
float per;
}
std1={101,655,89.90};
cout<<“roll=”<<roll);
cout<<“mark of student=”<<mark);
cout<<“percentage=”<<per;
}
```

Note the following points while declaring a structure type:

1) The closing brace in the structure type declaration must be followed by a semicolon.
2) It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the ‘form’ of the structure.
3) Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined.

Rajesh S. Jha
Lecturer (Modern College)

Accessing Structure Elements

In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. So to refer to **pages** of the structure defined in our sample program we have to use,

b1.pages

Similarly, to refer to **price** we would use,

b1.price

Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

Array of structure:

As we know that structure are used to describe the formats of a number of logically related variables. If there are number of variable declares in structure then it is not convenient to use different variable.

For ex if we want to store data of 1000 books then we required 1000 different structure variables which are not convenient. So better approach to use an array of structure.

```
/*WAP a program to find book details using array of structure*/
#include<stdio.h>
#include<conio.h>
main()
{
int I;
struct lib1
{
char bname[20];
int price;
int page;
}book[3];
clrscr();
cout<<“enter the details of three book”;
for(i=0;i<3;i++)
{
cout<<“\nenter the name of book”;
cin>>book[i].bname;
cout<<“\nenter the price of book”;
cin>>book[i].price;
cout<<“\nenter the no of page in book”;
cin>>“%s”,&book[i].page;
}
```

```
cout<<"book details:";

for(i=0;i<3;i++)
{
cout<<"\n the name of book\t"<<book[i].bname;
cout<<"\n the price of book\t"<<book[i].price;
cout<<"\n the no of page in book"<<book[i].page;
}
getch();
return(0);
}
```

Union:

Union is similar to structure. Like structure union contain members, whose individuals data type may be differ from one another. In structure, each member has its own memory location, whereas member of union share the same memory location. We can assign values to only one member at a time, so assigning value to another member that time has no meaning.
When union is declare, compiler automatically allocates memory locations to hold the largest data type of member in the union. Thus union is used for saving memory.

Syntax:

```
union  union_name
{
Member 1;
-----
Member n;
}variable_name;
```

```
/* program for accessing union member*/
main()
{
union result
{
int marks;
char grade;
}res;
res.marks=90;
cout<<" marks:"<<res.marks;    //where grade has no meaning
cout<<" grade="<<res.grade;
res.grade='A';
cout<<" marks:"<< res.marks;   //where mark has no meaning
cout<< "grade="<< res.grade;
}
}
```

Difference between structure and union:

Structure	Union
Structure is a user defined data type and it is used to group a number of different variables together.	Union is a user defined data type and it is used to group a number of different variables together.
Structure variables hold different memory location for each member.	Union variables hold same memory location for each member.
Structure members are strictly treated as a same type as defined in structure	Union offers a way for a section of memory to be treated as a variable of one type on one occasion, and different variable of a different type on another occasion.
Structure consume more memory	Union does not consume more memory
We can store value of all members at a time.	We can store value of only one member at a time.
Several members of a structure can be initialized at once.	Only the first members of a structure can be initialized at once.
Value of variable is predictable	Value of variable is not predictable.

Passing Structures to Functions:

When we pass a member of a structure to a function, we are actually passing the value of that member to the function. Therefore, we are passing a simple variable.
For example, consider this structure:

```
struct fred
{
char x;
int y;
float z;
char s[10];
} mike;
```

Here are examples of each member being passed to a function:

```
func(mike.x);          /* passes character value of x */
func2(mike.y);              /* passes integer value of y */
func3(mike.z);              /* passes float value of z */
func4(mike.s);              /* passes address of string s */
func(mike.s[2]);       /* passes character value of s[2] */
```

If we wish to pass the *address* of an individual structure member, put the **&** operator before the structure name.
For example, to pass the address of the members of the structure **mike**, write

```
func(&mike.x);              /* passes address of character x */
```

```
func2(&mike.y);        /* passes address of integer y */
func3(&mike.z);        /* passes address of float z */
func4(mike.s);                 /* passes address of string s */
func(&mike.s[2]);      /* passes address of character s[2] */
```

Remember that, & operator precedes the structure name, not the individual member name. Note also that **s** already signifies an address, so no **&** is required.

Passing Entire Structures to Functions:

When a structure is used as an argument to a function, the entire structure is passed using the standard call-by-value method. Of course, this means that any changes made to the contents of the structure inside the function to which it is passed do not affect the structure used as an argument.

When using a structure as a parameter, remember that the type of the argument must match the type of the parameter. For example, in the following program both the argument **arg** and the parameter **parm** are declared as the same type of structure.

```
#include <stdio.h>
/* Define a structure type. */
struct struct_type {
int a, b;
char ch;
} ;
void f1(struct struct_type parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg);
return 0;
}
void f1(struct struct_type parm)
{
printf("%d", parm.a);
}
```

As this program illustrates, if we will be declaring parameters that are structures, we must make the declaration of the structure type global so that all parts of wer program can use it.

For example, had **struct_type** been declared inside **main()** (for example), then it would not have been visible to **f1().**

As just stated, when passing structures, the type of the argument must match the type of the parameter. It is not sufficient for them to simply be physically similar; their type names must match.

For example, the following version of the preceding program is incorrect and will not compile because the type name of the argument used to call **f1()** differs from the type name of its parameter.

```
/* This program is incorrect and will not compile. */
#include <stdio.h>
/* Define a structure type. */
struct struct_type {
int a, b;
char ch;
} ;


/* Define a structure similar to struct_type,
but with a different name. */
struct struct_type2 {
int a, b;
char ch;
} ;
void f1(struct struct_type2 parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg); /* type mismatch */
return 0;
}
void f1(struct struct_type2 parm)
{
printf("%d", parm.a);
}
```

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
Enumerations are defined much like structures; the keyword **enum** signals the start of an enumeration type. 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;
```

Classes and object:

A class is a way to bind the data and its associated function together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type i.e. user defined data type that can be treated like any other built in data type.

Generally class specification has two parts:

1. Class declaration
2. Class function definitions

The **class declaration** describes the type and scope of its members. The **class function definitions** described how the class functions are implemented.

General form of class declaration is:

```
class class_name
{
      private:
            Variable declaration;
            Function declaration;
      public:
             Variable declaration;
             Function declaration;

};
```

The class declaration is similar to a **struct** declaration. The keyword **class** specifies that what follows in an abstract data of type class_name. The body of class is enclosed within braces and terminated by semicolon. The class body contains declaration of variable and functions. Variable and functions are known as class member.

A **class definition** consists of two parts: header and body. The class **header** specifies the class **name** and its **base classes**. The class **body** defines the class **members**.

Two types of members are supported:

- **Data members** have the syntax of variable definitions and specify the representation of class objects.
- **Member functions** have the syntax of function prototypes and specify the class operations, also called the class **interface**.

Class members are usually grouped into three sections:

1. **Private**
2. **Public**
3. **Protected**

These keywords are known as visibility labels.

Private: The class member that has been declaring as private can be accessed only from within a class. On the other hand public member can be access from out side the class also.
The data hiding using private declaration is a key feature of object oriented programming. The use of keyword private is optional.
By default, the members of class are private. If we not declare access of class member then it is private member. **Private** members are only accessible by the class members.

Public: The **public** access specifier allows functions or data to be accessible to other parts of our program. **Public** members are accessible by all class users.

Protected: The **protected** access specifier is needed only when inheritance is involved.
Protected member of class can be access from only its child class. **Protected** members are only accessible by the class members and the members of a derived class.

Ex:

```
class item
{
        int number1;    // variable declaration private by default
        float cost;
    public:
        void getdata(int a, float b);     //function declaration using prototype and it is public
        void putdata(void);
};      //end with semicolon
```

Creating object:

We define an object of wer new type just as we define an integer variable: An *object* is an individual instance of a class.
Ex:

```
int GrossWeight; // define an integer
Cat Frisky; // define a Cat
item Item1;//define a item
```

This code defines a variable called GrossWeight whose type is an integer. It also defines Frisky, which is an object whose class (or type) is Cat. It also defines item1, which is an object whose class (or type) is item.

Accessing class member:

We can access class member by using following sntax:

```
Objectname.function_nmae(actual arguments)
```

Ex:

In above program i.e. in class item we can define an object in following way:

```
item x; //memory for x is created
x.getdata(100,75.5);
x.putdata();
```

```
ex:
class xyz
{
                int x;
                int y;
        public:
                int z;
};
------
------
Xyz p;
p.x=10;//error zx is private
p.z=20;//ok, z is public
```

Defining member function:

Member function can be defined at two place:

1. Outside the class definition
2. Inside the class definition

Outside class definition:

Member functions that are declared inside a class have to be defined separately outside the class. Their definition very much similar like normal functions. They should have a function header and a function body.

The importance difference between a member function and a normal function is that a member function required 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 outside the class is:

```
Return_type class_name::function_nmae(argument declaration)
{
Function body;
}
```

Ex:

```
Void item::getdata(int a, float b)
{
Number=a;
Cost=b;
}
Void item::putdata(void)
{
cout<<"Number"<<number<<endl;
cout<<"cost"<<cost;
}
```

Inside the class definition:

Member functions that are declared inside a class have to be defined at the same place inside the class. Their definition very much similar like normal functions. They should have a function header and a function body.

The general form of a member function definition outside the class is:

```
public/private:
return_type function_nmae(argument declaration)
{
Function body;
}
```

Ex:

```
Class item
{
        int number;
        float cost;
public:
        void getdata(int a, float b)
        {
        number=a;
        cost=b;
        }
        void putdata(void)
        {
        cout<<"Number"<<number<<endl;
```

```
        cout<<"cost"<<cost;
        }
};
```

Defining Inline member function:

Inline Functions:

In C++, we can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro.

To cause a function to be expanded in line rather than called, precede its definition with the **inline** keyword.

For example, in this program, the function **max()** is expanded in line instead of called:

```
#include <iostream>
using namespace std;
inline int max(int a, int b)
{
return a>b ? a : b;
}
int main()
{
cout << max(10, 20);
cout << " " << max(99, 88);
return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

```
#include <iostream>
using namespace std;
int main()
{
cout << (10>20 ? 10 : 20);
cout << " " << (99>88 ? 99 : 88);
return 0;
}
```

The reason that **inline** functions are an important addition to C++ is that they allow we to create very efficient code.

Defining Inline Functions within a Class:

It is possible to define short functions completely within a class declaration. When a function is defined inside a class declaration, it is automatically made into an **inline** function (if possible).

It is not necessary (but not an error) to precede its declaration with the **inline** keyword.

For example, the preceding program is rewritten here with the definitions of **init()** and **show()** contained within the declaration of **myclass**:

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
// automatic inline
void init(int i, int j) { a=i; b=j; }
void show() { cout << a << " " << b << "\n"; }
};
int main()
{
myclass x;
x.init(10, 20);
x.show();
return 0;
      }
```

Defining Inline Functions outside a Class:

Inline functions may be class member functions. If we defined inline function outside the class it is necessary to add keyword inline as a header before function.

For example, this is a perfectly valid C++ program:

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
void init(int i, int j);
void show();
};
// Create an inline function.
inline void myclass::init(int i, int j)
{
a = i;
b = j;
}

// Create another inline function.
inline void myclass::show()
{
```

```
cout << a << " " << b << "\n";
}
int main()
{
myclass x;
x.init(10, 20);
x.show();
return 0;
}
```

Static Class Members:

Both function and data members of a class can be made **static**.

Static Data Members:

The data members of the class which do not require object of the class for initialization or execution are known as static member. Where the member of class which require object of the class for initialization or execution are known as instance member.
A variable declare using static keyword or under static block was static variable, rest all are instance variable.

```
Ex:
Int x=100;   // instance member
Static int y=100;   //ststic member
Static void main()
{
Int x=100;  //static member
}
```

Static variable gets initialized once the execution of the class starts whereas instance variable gets initialized only after the object of the class created.

The minimum and maximum number of time the static member gets initialized is one and only one whereas in case of instance it will be zero or n times.

To invoke static member we required class_name, whereas to invoke instance member we used object of the class.

In sort we can say that, When we precede a member variable's declaration with **static**, we are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. All **static** variables are initialized to zero before the first object is created.

To understand the usage and effect of a **static** data member, consider this program:

```
#include <iostream>
using namespace std;
class shared
{
   static int a;
    int b;
public:
   void set(int i, int j)
   {a=i; b=j;}
   void show();
} ;
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
```

The integer **a** is declared both inside **shared** and outside of it. Because this is necessary because the declaration of **a** inside **shared** does not allocate storage.

Static Member Functions:

Member function declares using by static keyword is a static member function rest of all is instance member function.

Static member function gets initialized once the execution of the class starts whereas instance member function gets initialized only after the object of the class created.

The minimum and maximum number of time the static member function gets initialized is one and only one whereas in case of instance it will be zero or n times.

To invoke static member function we required class_name, whereas to invoke instance member function we used object of the class.

A static function can have access to only other static members (functions or variables) declared in the same class.

Inside static function, we can not consumed directly non static member of the class directly. It can be done by using object of the class.

In the following example we describe these concepts.

```
Ex:
class statmet
{
int x=100;
static int y=50;//static variable
int r;
static void add() //static function
{
statmet obj;   //create object
r=obj.x+y;
}
};
void main()
{
statmet::add();//using class name for calling static function
getch();
}
```

Array within a class:

The array can be used as member variables in a class. The following class definition is valid:

```
const int size=10;//provide value for array size
class array
{
int a[size]; // a is int type array
public:
void stval(void);
void display(void);
}
```

The array variable a[] declared as private member of the class array can be used in the member functions, like any other array variable. We can perform any operation on it.
For example, in the above class definition, the member function setval() sets the values of elements of the array a[], and display() function display the values. Similarly, we may use other member functions to perform any other operation on the array values.

Array of objects:

As we know that an array can be of any data type including struct. Similarly, we can also have array of variables that are of the type class. Such variables are called array of objects.
Consider following class definition:

```
class employee
{
char name[30];
float age;
public:
void getdata();
void putdata();
};
```

The identifier employee is a user defined data type and can be used to create objects that relate to different categories of the employees.
Ex:

```
employee manager[3];   //array of manager
employee foreman[13];   //array of foreman
employee worker[75];   //array of worker
```

The array manager contains three objects (managers), namely, manager[0], manager[1], manager[2], of type employee class, similarly, the foreman array contains 15 objects (foreman) and the worker array contains 75 objects (workers).
Since an array of objects behave 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.
For example:

```
Manager[i].putdata();
```

Will display the data of the ith elements of the array manager. That is, this statement requests the object manager[i] to invoke the member function putdata().

An array of object is stored inside the memory in the same way as a multi-dimensional array.

The array manager represented in figure:

name
age
name
age
name
age

Note that only the space for data items of the objects is created. Member function are store separately and will be used by all the objects.

```
#include<iostream.h>
Using namespace std;
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;
int main()
{
employee manager[size];
for(int i=0;i<size;i++)
{
cout<<”detail of manager”;
manager[i].getdata();
}
```

```
cout<<"\n";
for(int i=0;i<size;i++)
{
cout<<"detail of manager"<<i+1<<"\n";
manager[i].putdata();
}
return 0;
}
```

Object as function arguments:

Object can be used as function arguments. This can be done in two ways:

- A copy of entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first way is call as **pass-by-value** and second way is call as **pass-by-reference**.

Following example illustrates the use of objects as function arguments. It performs the addition of time in hour and minutes format:

```
# include<iostream.h>
using namespace std;
class time
{
     int hours;
     int minutes;
public:
   void gettime (int h, int m)
   {
     hours=h;
     minutes=m;
   }
   void putdtime(void)
   {
   cout<<hours<<"hours and";
    cout<<<<minutes<<"minutes"<<"\n";
   }
void sum(time, time);  //declaration with objects as arguments
};
void time::sum(time t1, time t2) // t1, t2 are objects
{
minutes=t1.minutes+t2.minutes;
```

```
hours=minutes/60;
minutes=minutes%60;
hour=hour+t1.hour+t2.hours;
}
int main()
{
time T1, T2, T3;
T1.gettime(2,45);//get T1
T2.gettime(3,30);//get T2
T3.sum(t1,t2);  //T3=T1+T2
cout<<"T1=";T1.puttime();  //Display T1
cout<<"T2=";T2.puttime();  //Display T2
cout<<"T3=";T3.puttime();  //Display T3
return 0;
}
```

Output:
T1=2 hour and 45 minutes
T2=3 hour and 30 minutes
T3=6 hour and 15 minutes

1) Object as function argument

```
class complex
{
      int x;
      int y;

public :
       complex(int x, int y)
       {
                this->x=x;
                this->y=y;
       }

       void display(complex c)
       {
            cout<<c.x<<" "<<c.y<<endl;
       }
};

int main()
{
    complex c1(10,5);
    c1.display(c1);

    cin.get();
    return 0;
```

```
}
```

For understanding purpose we used this matter, not foe exam purpose:

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-by value mechanism. This means that a copy of an object is made when it is passed to a function.

However, the fact that a copy is created means, in essence, that another object is created. This raises the question of whether the object's constructor function is executed when the copy is made and whether the destructor function is executed when the copy is destroyed.

The answer to these two questions may surprise we. To begin, here is an example:

```
// Passing an object to a function.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
myclass(int n);
~myclass();
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass::myclass(int n)
{
i = n;
cout << "Constructing " << i << "\n";
}
myclass::~myclass()
{
cout << "Destroying " << i << "\n";
}
void f(myclass ob);
int main()
{
myclass o(1);
f(o);
cout << "This is i in main: ";
cout << o.get_i() << "\n";
return 0;
}
void f(myclass ob)
{
```

```
ob.set_i(2);
cout << "This is local i: " << ob.get_i();
cout << "\n";
}
```

This program produces this output:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

Notice that two calls to the destructor function are executed, but only one call is made to the constructor function. As the output illustrates, the constructor function is not called when the copy of **o** (in **main()**) is passed to **ob** (within **f()**).

The reason that the constructor function is not called when the copy of the object is made is easy to understand. When we pass an object to a function, we want the current state of that object.

If the constructor is called when the copy is created, initialization will occur, possibly changing the object. Thus, the constructor function cannot be executed when the copy of an object is generated in a function call.

Returning object:

A function cannot only receive objects as arguments but also can return them. A function may return an object to the caller. For example, this is a valid C++ program:

```
// Returning objects from a function.
#include <iostream>
using namespace std;
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();
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.

There are ways to overcome this problem that involve overloading the assignment operator and defining a copy constructor.

-----End of 3rd unit---

Constructor and destructor:

"A constructor is a special member function of a class. Its task is to initialize the object of the class." Whenever object is created, special member function i.e. constructor will be executed automatically. Constructor function is different from all other member function in a class because it is used to initialized variables. It is called constructor because it construct the values of data member of the class.

Syntax rule for writing constructor function:

a) A constructor name must be the same as that of its class name.
b) It is declare with no return type. Because it does not return anything's.
c) Constructor may not be static or virtual.
d) It should be public. Only in some rare case it can be private.

The general syntax of constructor is:

```
Class class_name
{
        Private:
                ……….
                ……….
        Public:
                Class_name();           //constructor declared
                ……..
                ……
};
Class_name::class_name()        //constructor defined
{
……
……..
……
}
```

Ex: We are going to implement CRectangle including a constructor:

```
// example: class constructor
#include <iostream>
using namespace std;
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;
}
```

o/p:
rect area: 12
rectb area: 30

Characteristics of constructor function:

1. The Constructor name is always same as the class name.
2. Constructor does not have return type, not even void and therefore, they can not return values.
3. They cannot be static or virtual.
4. They should be declared in public section.
5. They can not inherit though a derived class can call only base class constructor.
6. Like c++ function, they can have default arguments.
7. We can not refer their address.
8. When a constructor is declared for a class, initialization of class objects become mandatory, since constructor is invoked automatically when the objects are created.
9. Constructor makes implicit call to operators 'new'.

Parameterized constructor:

It is possible in c++ to pass arguments to the constructor function when the objects are created. These constructors are called parameterized constructor.
Ex:

```
Class integer
{
        int m,n;
        public:
                integer(int x, int y);     //parametrised constructor
                ……..
                ……..
};

Integer::integer(int x, int y)
{
m=x;
n=y;
}
```

When constructor has been parameterized, the objects declaration statement such as
integer int1;
May not work.
We must pass initial value to the constructor as arguments, when an object is declared.
This can be done in two ways:

1. By calling constructor explicitly
2. By calling the constructor implicitly.

```
integer int1=integer(12,23);    //explicit call
integer int1(12, 23);    //implicit call
```

Ex: Example of parameterized constructor:

```
#include <iostream>
#include <cstdio>
using namespace std;
class date
{
int day, month, year;
public:
date(int m, int d, int y);  // parameterized constructor with two arguments
void show_date();
};
// Initialize using integers.
date::date(int m, int d, int y)
{
day = d;
month = m;
year = y;
}
void date::show_date()
{
cout << month << "/" << day;
cout << "/" << year << "\n";
}
int main()
{
date ob1(12, 4, 2001);  // invoking constructor at the time of object creation
ob1.show_date();
return 0;
}
```

Multiple constructors in a class (Overloading 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>
using namespace std;
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
```

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.

Constructor with default arguments:

It is possible to defined constructor with default arguments.
For example:
The constructor complex() can be declared as:

Complex(float real, float imag=0);

The default value of the arguments imag is zero.
The statements complex(5.0);

Assigns the value 5.0 to the real and 0.0 to imag (by default)

However the statements complex(2.0,3.0);

Assigns the value 2.0 to the real and 3.0 to imag .

The actual parameter, when specified, overrides the default value. The missing arguments, must be trailing ones.

Default constructor:

The default constructor is different from constructor with default argument. The default constructor is special member function invoked by the C++ compiler without any arguments for initializing the objects of a class.
For example:
A::A(); is default constructor for class A.

Rajesh S. Jha
Lecturer (Modern College)

Dynamic initialization of objects:

Class objects can be initialized dynamically i.e. the initial value of an object may be provided during runtime. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different data formats at runtime.

The following example illustrates this:

Dynamic initialization of constructor:

```
#include<iostream.h>
class fixed_deposite
{
long int p_ampount;
int years;
float rate;
float r_value;
        public:
                fixed_deposit(){};
                fixed_deposit(long int p,int y,float=0.12);
                fixed_deposit(long int p,int y,int r);
                void display();
};
fixed_deposit::fixed_deposit(long int p,int y,int r);
{
p_ampount=p;
years=y;
 rate=r;
r_value=p_amount;
        for(int i=1;i<=y;i++)
        r_value=r_alue*(1.0+r);
}

fixed_deposit::fixed_deposit(long int p,int y,int r);
{
p_ampount=p;
years=y;
 rate=r;
r_value=p_amount;
        for(int i=1;i<=y;i++)
        r_value=r_alue*(1.0+ float(r)/100);
}
void fixed_deposit::display()
{
cout<<"\n"<<"principal amount="<<p_amount<<"\n"<<"return value="<<r_value<<"\n";
```

```
}
void main()
{
fixed_deposit fd1,fd2,fd3;
long int p;
int y;
float r;
int R;
cout<<"enter amount,periods and interest rate(percent)"<<"\n";
cin>>p>>y>>R;
fd1=fixed_deposit(p,y,R);

cout<<"enter amount,periods and interest rate(decimal)"<<"\n";
cin>>p>>y>>r;
fd1=fixed_deposit(p,y,r);

cout<<"enter amount,periods"<<"\n";
cin>>p>>y;
fd1=fixed_deposit(p,y);

cout<<"\n deposit 1";
fd1.display();

cout<<"\n deposit 2";
fd2.display();

cout<<"\n deposit 3";
fd3.display();
}
```

In the above program we use three overloaded constructors.
The parameter value is provided at runtime. The user can provide input in one of the following forms:

- **Amount, period, interest(decimal form)**
- **Amount, period, interest(percent form)**
- **Amount and periods**

The 2nd constructor is invoked for form 1 and 3. Third is invoked for form 2. For form 3, it used default value of r.

Copy constructor:

Copy constructor always used when compiler has to create a temporary object of a class object. The copy constructor used in following situations:

1. The initialization of an object by another object of the same class.

2. Return of object as by value parameters of a functions
3. Starting the objects as by value parameters of a functions.

The general formats of copy constructor are:

Class_name::class_name(class_name &ptr)

Ex:

x::x(x& ptr)

ptr is a pointer to a class object X

The statements x i2(i1);

Would define objects i2 and at the same time the initialization it to the values of i1.

The following program show how define copy constructor:

```
fib::fib()  //constructor
{
f0=0;
f1=1;
f=f0+f1
}
fib::fib(fib & ptr)  //copy constructor
{
f0=ptr.f0;
f1=ptr.f1;
f=ptr.f;
}
```

Destructor:

The destructor is a function which automatically executes when objects is destroyed. A destructor function gets executed whenever an instance of the class to which it belongs goes out of existence

The primary usages of destructor function are to replace space on heap. A destructor function may be invoked explicitly.

Syntax rule for writing destructor function:

1. A destructor function name is same as that of the class it belongs accepts that the first character of the name must be a tidle (~).
2. It is declare with no return type since it can not ever return a value.
3. It takes no arguments.
4. It should have public access in the class declaration.

The general syntax:

```
class user_name
{
……..
public:
user_name();     //constructor
~user_name();   //distructor
}
```

Ex:

```
// example on constructors and destructors
#include <iostream>
using namespace std;
class CRectangle {
int *width, *height;
public:
CRectangle (int,int);
~CRectangle ();
int area () {return (*width * *height);}
};
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;
}
```

o/p:

```
rect area: 12
rectb area: 30
```

Rajesh S. Jha
Lecturer (Modern College)

Operator overloading and type conversion:

Operator overloading is an important feature of c++ language. In c++ the user defined data types can behave in much the same way as built in data 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 ability to provide the operator with special meaning for a data type. The mechanism of giving such special meaning to an operator is known as operator overloading.
Overloading stands for giving additional meaning to operator. We can overload all operators except the following:

1. Class member access operator (. , *)
2. Scope resolution operator (::)
3. Sizeof operator
4. Conditional operator (?,:)

When operator is overload its original meaning is not lost. For example, which is overloaded to add two vectors, can still be used to add two integers.

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 special kind of function, called operator function. The general form of operator function is:

```
return_type class_name::operator op_to_be_overloaded(arg_list)
{
Function body   // task defined
}
```

Where retun_type is the type of value returned by the specified operation. The operator to be overloaded is preceded by the keyword operator.
The operator function is either, **member function** and **friend function**. The friend functions have one argument for unary operators and two for binary operators. The member functions have no argument for unary operators and only one for binary operators.
Only those operators that are predefined in c++ compiler are allowed to be overloaded.

Following are some example of operator overloading of functions:

Void operator ++(); is equal to void increment();
Void operator + (int x, int y); is equal to int sum(int x, int y)

The process of overloading involves following steps:

1. First create a class that defines the data_type that is to be used in overloading operation.
2. Declare operator function in public part of class. It may be either **member function** or **friend function.**

3. Define the operator function to implements the required operation.
4. For binary operator, overloaded operator function can be invoked by the expression such as:
   x op y
5. For friend function op x written as operator op(x) and x op y as x operator op(y) or operator op (x,y)
6. For unary operator, overloaded operator function can be invoked by the expression such as:
   op x or x op.

Rules of overloading operator:

1. Only existing operator can be overloaded. New operator can not be created.
2. The overloaded operator must have at least one operand that is user defined type.
3. We can not change the basic meaning of operator.
4. Overloaded operator followed the syntax rule of the original operators.
5. Certain operator can not be overloaded.

Sizeof	Sizeof operator
.	Membership operator
.*	Pointer to member operator
::	Scope resolution operator
?:	Conditional operator

6. Certain operator can not be overloaded by friend function.

=	Assignment operator
()	Function call operator
[]	Subscripting operator
->	Class member access operator

7. The friend functions have one argument for unary operators and two for binary operators.
8. The member functions have no argument for unary operators and only one for binary operators.
9. Operators ++ and -- can be overloaded as prefix as well as postfix.

Example:

1. **Overloading unary operator**
   Write a program in c++ to overloaded unary operator, so that the unary minus operator applied to an object should change the sign of each of its data items.

```
#include <iostream.h>
class space
{
int x, y, z;
        public:
        void getdata(int a, int b, int c);
        void display(void);
        void operator –();       //operator overloading
};
void space::getdata(int a, int b, int c)
{
x=a;
y=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;      //activates operator –()
cout<<”s:”;
s.display();
}
```

o/p:
s: 10 -20 30
s: -10 20 -30

2. **Overloading binary operator**

Rajesh S. Jha
Lecturer (Modern College)

Write a program in c++ to overloaded binary operator + for addition of complex number.

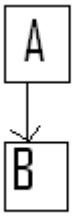
```
#include<iostream.h>
class complex
{
float x; //real part
float y;  //imaginary part
public:
        complex(){}    //constructor1
        complex(float real, float imag)          //constructor2
        {
        x=real;
        y=imag;
        }
        complex operator +(complex);
void display(void);
};
complex complex::operator +(complex c)
{
complex temp;
temp.x=x+c.x;
temp.y=y+c.y;
return(temp);
}
void complex::display(void)
{
cout<<x<<"+j"<<y<<"\n";
}
main()
{
Complex c1, c2, c3;   //invokes constructor1
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();
}
o/p:
c1=2.5+j3.5
c2=1.6+j2.7
c3=4.1+j6.2
```

Rajesh S. Jha
Lecturer (Modern College)

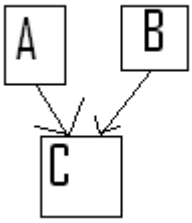
Inheritance:

Inheritance stands for derivation. The mechanism of deriving a new class from an old class is called inheritance. The old class is known as base class and new class is known as derived class.

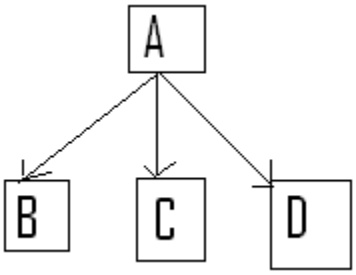
A derived class with only one base class is known as **single inheritance.**



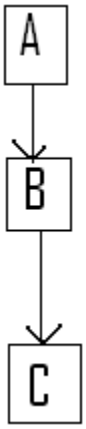
A derived class with several base classes is known as **multiple inheritances.**



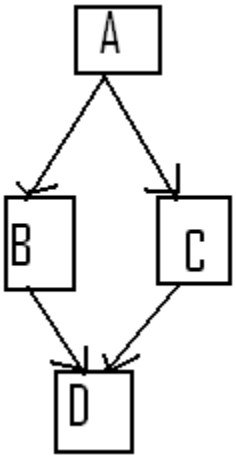
When more than one class is derived from one base class then it is **hierarchical inheritance.**



The mechanism of deriving a class from another derived class is known as **multilevel inheritance.**



The **hybrid inheritance** is shown below:



Inheritance in c++ allows us to reuse the class that has been tested. It saves the effort of developing and testing the same again. It not only saves time and money but also reduce frustration of increase reliability.

Defining derived class:

A derived class is defined by specifying its relationship with base class. The general form is shown in below:

```
class derived_class_name:visibility_mode base_class_name
{
……….
……..
//member of derived class.
}
```

The colon indicates that the derived_class_name is derived from base_class_name. The visibility mode can be either private, public, or protected. It is optional. The default visibility mode is private. Visibility mode specifies that whether the member of base class publicly derived or privately derived.
For example:

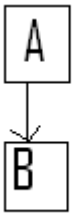
```
class abc: private xyz  //private derivation
{
Member of abc;
};

class abc: public xyz   //publicly derivation
{
Member of abc;
};

class abc: xyz  //private derivation by default
{
Member of abc;
};
```

Single inheritance:

A derived class with only one base class is known as single inheritance.



Ex:

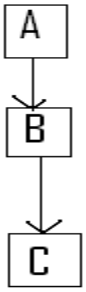
```
// derived classes
#include <iostream>
using namespace std;
class CPolygon
{
protected:
int width, height;
public:
     void set_values (int a, int b)
    {
     width=a; height=b;
     }
};
class CRectangle: public CPolygon
{
public:
     int area ()
     {
      return (width * height);
      }
};
int main ()
{
CRectangle rect;
rect.set_values (4,5);
cout << rect.area() << endl;
return 0;
}
0/p:
20
```

The objects of the classes CRectangle and CTriangle each contain members inherited from CPolygon. These are:
width, height and set_values().
These are the example of single inheritance. Becaused both the class derived from same base class.

Multilevel inheritance:

The mechanism of deriving a class from another derived class is known as multilevel inheritance.

Rajesh S. Jha
Lecturer (Modern College)



Ex: example of our family.

```
// derived classes
#include <iostream>
using namespace std;
class CPolygon
{
protected:
int width, height;
public:
        void set_values (int a, int b)
        {
        width=a;
         height=b;
        }
};
class CRectangle: public CPolygon
{
        public:
        int area1 ()
        {
        return (width * height);
        }
};
class CTriangle: public CRectangle
{
        public:
        int area ()
        {
        return (width * height / 2);
         }
};
int main ()
{
CTriangle trgl;
trgl.set_values (4,5);
```

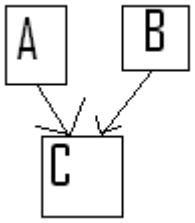
```
cout << trgl.area1() << endl;
cout << trgl.area() << endl;
return 0;
}
```

o/p:
20
10

In above example Ctriangle class derived from CRectangle class and CReactangle class derived from CPolygon. Hence this is a example of multilevel inheritance.

Multiple inheritances:

A derived class with several base classes is known as multiple inheritances.



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. 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>
using namespace std;
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;
}
```

Constructor and destructor in derived class:

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 we want that an overloaded constructor is called when a new derived object is created, we can specify it in each constructor definition of the derived class:

Rajesh S. Jha
Lecturer (Modern College)
Syntax:
derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

For example:

```
// constructors and derived classes
#include <iostream>
using namespace std;
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

Rajesh S. Jha
Lecturer (Modern College)

Rajesh S. Jha
Lecturer (Modern College)