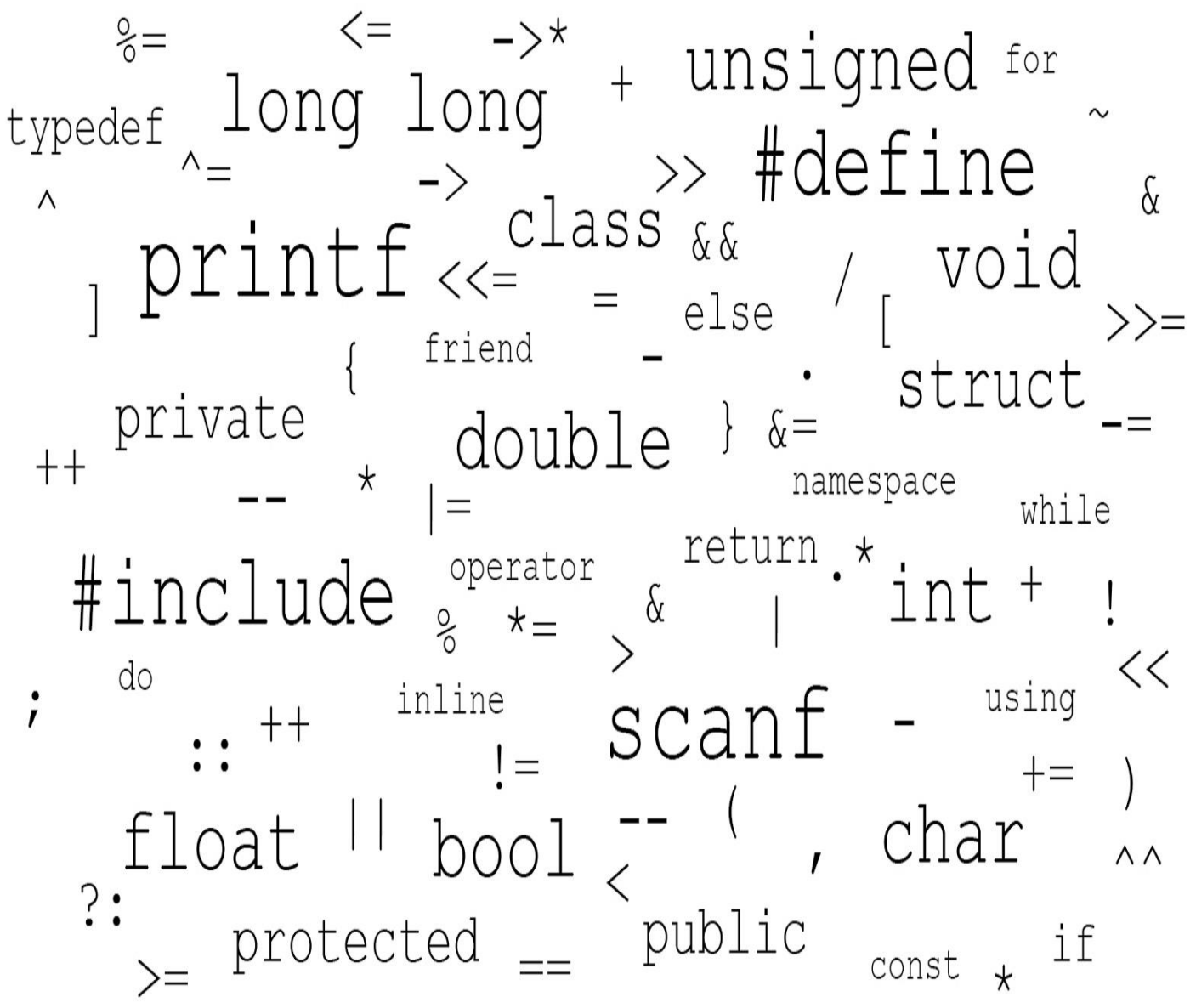


MODULE -1

BEGINNING WITH C++ AND ITS FEATURES

GANESH Y
Dept. of ECE RNSIT

MODULE -1
Beginning with C++ and its features

SYLLABUS

Beginning with C++ and its features: What is C++, Applications and structure of C++ program, Different Data types, Variables, Different Operators, expressions, operator overloading and control structures in C++ (Topics from Chapter-2,3 of Text1).

Differences between POP and OOP

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are: C, VB, FORTRAN, Pascal.	Example of OOP are: C++, JAVA, VB.NET, C#.NET.

Objects:-

* *Objects* are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

* They may also represent user-defined data such as vectors, time and lists, Programming problem is analyzed in terms of objects and the nature of communication between them.

* Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal or a structure in C.

Classes:-

* We just mentioned th.at objects contain data, and code *to* manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

* In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created.

* A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language.

* The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit bas been defined as a class, then the statement

```
fruit mango;
```

will create an object **mango** belonging to the class **fruit.**

Data Encapsulation: -

* The wrapping up of data and functions into a single unit (called class) *is* known as ***encapsulation****.* Data encapsulation is the most striking feature of a class.

* The data is not accessible to the out-side world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program *is* called ***data hiding*** *or* ***information hiding.***

Data Abstraction:-

* *Abstraction* refers to the act of representing essential features without including the background details or explanations.

* Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost and *functions* to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called *data numbers* because they hold information.

The functions that operate on these data are sometimes called ***methods or member functions.***

* Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

Inheritance:-

* *Inheritance* is the process by which objects of one class acquire the properties of objects of another class**.** It supports the concept of ***hierarchical classification.***

* In OOP, the concept of inheritance provides the idea of *reusability.* This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.

Polymorphism :-

* *Polymorphism* is another important OOP concept. Polymorphism a Greek term, means the ability to take more than one form.

* An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings. then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as ***operator overloading****.*

* Similarly Using a single function name to perform different types of tasks is known as ***function overloading****.*

Benefits of OOP

• Through inheritance, we can eliminate redundant code and extend the use of existing classes.

• We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving *of* development time and higher productivity.

• The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

• It is possible to have multiple instances of an object to co-exist without any interference.

• It is possible to map objects in the problem domain to those in the program.

• It is easy to partition the work in a project based on objects.

• The data-centered design approach enables us to capture more details of a model in implementable form.

• Object-oriented systems can be easily upgraded from small to large systems.

• Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.

• Software complexity can be easily managed.

What is C++?

* **C++ is an object-oriented programming** language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in USA.
* C++ is an extension of C with a major addition of the class construct feature of Simula67.
* Since the class was a major addition to the original C language, Stroustrup initially called the new language **'C with classes'.** However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.
* During the early 1990's the language underwent a number of improvements and changes. In November 1997, the ANSI/ISO standards committee standardized these changes and added several new features to the language specifications.
* C++ is a superset of C. Most of what we already know about C applies to C++ also. Therefore. almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler.
* The most important facilities that C++ adds on to C arc classes. inheritance, function overloading, and operator overloading. Those features enable creating of abstract data types, inherit properties &om existing data types and support polymorphism, thereby making C++ a truly object-oriented language.
* The addition of new features has transformed C from a language that currently facilitates top-down. structured design, to one that provides bottom-up, object oriented design.

Applications of C++

C++ is a versatile language for handling very large Programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.

• Since C++ allows us to create hierarchy-related objects, we can build special object, oriented libraries which can be used later by many programmers.

• While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.

• C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object

A Simple C++ Program

Example of a C++ program that prints a string on the screen.

```
# include <iostream> // include header file
using namespace std;
int main ()
{
      cout<<"Hello world\n";   // C++ statement
      return 0;
}     // end of example
```

This simple program demonstrates several C++ features.

Program Features

Like C, the C++ program is a collection of functions. The above example contains only one function, `main()`. As usual, execution begins at `main()`.

Every C++ program must have a `main()`. C++ is a free-form language. With a few exceptions, the compiler ignores carriage returns and white spaces. Like C, the C++ statements terminate with semicolons.

Comments

```
// This is an example of
// C++ program to illustrate
// Some of its features
```

The double slash comment is basically a single line comment. Multiline comments can be written as follows:

```
/* This is an example of
C++ program to illustrate
Some of its features */
```

the double slash comment cannot be used in the manner as shown below:

```
for(j=0; j<n;/* loops n time*/ j++)
```

Output Operator

The only statement in above program is an output statement. The statement

```
cout <<"Hello world\n";
```

causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<.

The identifier cout (pronounced as 'C out') is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices.

The operator << is called the *insertion or put to* operator. It inserts (or sends) the contents of the variable on its right to the object on its left as shown in fig.1.

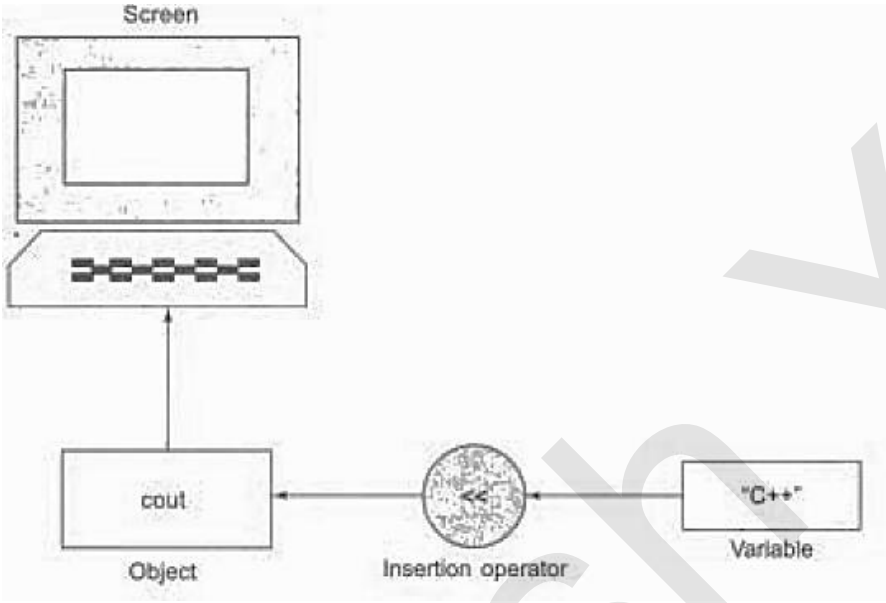


Fig.1 output using insertion operator

The iostream File

We have used the following #include directive in the program:

```
#include <iostream>
```

This directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier cout and the operator <<.

Some old versions of C++ use a header file called iostream.h. This is one of the changes introduced by ANSI C++.

Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the namespace scope we must include the using directive, like

```
using namespace std;
```

Here, **std** is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in **std** to the current global scope. **using** and **namespace** are the new keywords of C++.

Return type of main( )

ln C++, main( ) returns an integer type value to the operating system. Therefore, every main( ) in *C++* should end with a return 0 statement; otherwise a warning or error might occur.

Since main() returns an integer type value, return type for main() is explicitly specified as int. Note that the default return type for all functions in C++ is **int**.

Variables

```
float number1, number2, sum, average;
```

All variables must be declared before they are used in the program.

Input Operator

The statement

```
cin >> number1;
```

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier **cin** (pronounced 'C in ') is a predefined object in C++ that corresponds to the standard input stream.

The operator >> is known as *extraction or get from* operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right (Fig.2). This corresponds to the familiar scanf ( ) operation. Like <<, the operator >> can also be overloaded.

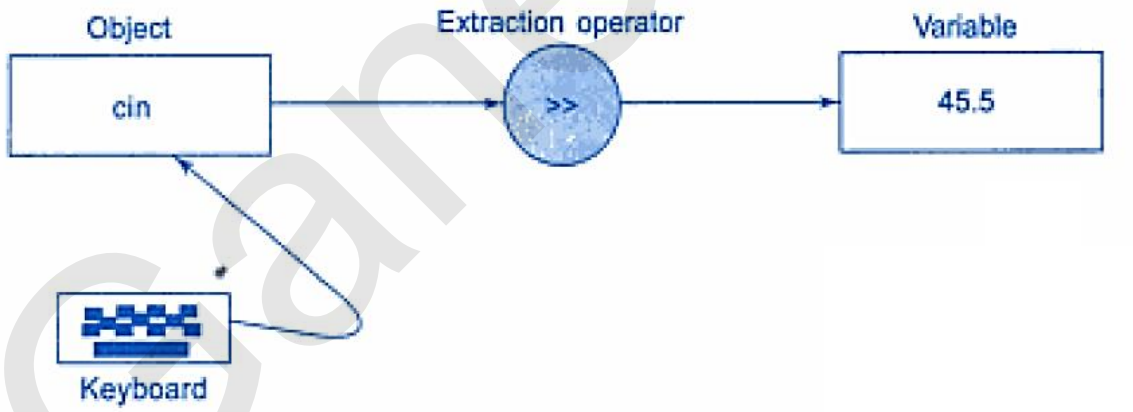


Fig.2 Input using extraction operator

Cascading of I/0 Operators

The statement

```
cout <<"Sum="<< sum<<"\n";
```

first sends the string "Sum=" to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line.

```
cout << "Sum=" << sum<<"\n"
     << "Average=" << average<<"\n";
```

This is one statement but provides two lines of output. If you want only one line of output, the statement will be:

```
cout << "Sum=" << sum<<","
     << "Average=" << average<<"\n";
```

We can also cascade input operator>> as shown below:

```
cin >>number1>> number2;
```

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to number1 and 20 to number2.

Structure of C++ Program

A typical C++ program would contain four sections as shown in Fig.3. These sections may be placed in separate code files and then compiled independently or jointly.

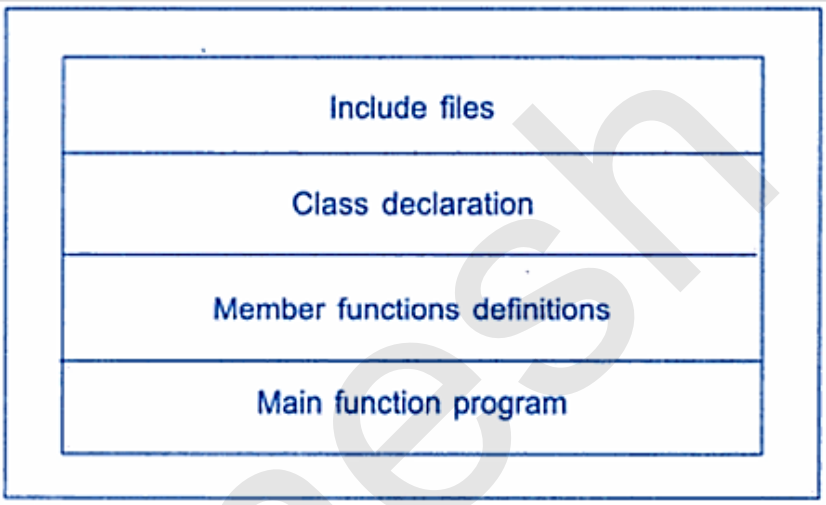


Fig 3 structure of a C++ program

It is a common practice to organize a program into three separate files:

* The class declarations are placed in a header file and the definitions of member functions go into another file.
* This approach enables the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member functions definition).
* Finally, the main program that uses the class is placed in a third file which "includes"
* the previous two files as well as any other files required.

This approach is based on the concept of client-server model as shown in Fig. 4. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

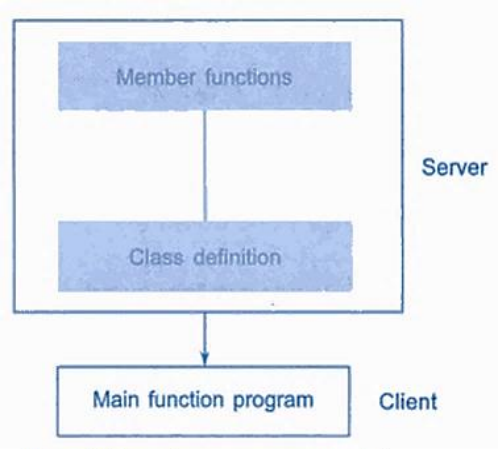


Fig. 4 The client-server model

Fallowing program shows the use of class in a C++ program.

```
#include <iostream>
using namespace std;
class person
{
     char name[30];
     int age;
     public:
     void getdata(void);
     void display(void);
} ;
void person :: getdata (void)
{
     cout << "Enter name: ";
     cin >> name;
     cout << "Enter age: ";
     cin >> age;
}
void person :: display(void)
{
     cout << "\n Name: " << name;
     cout << "\n Age:  " << age;
}
int main()
{
     person p;
     p.getdata();
     p.display();
     return 0;
}
```

The program defines person as a new data of type class. The class person includes two basic data type items and two functions to operate on that data. These functions are called **member** functions. The main program uses person to declare variables of its type. As pointed out earlier, class variables are known as **objects**. Here, **p** is an object of type **person**.

Tokens

The smallest individual units in a program are known as tokens. C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements. Table. 1 gives the complete set of C++ keywords.

Table 1 C++ keywords

asm	**double**	new	**switch**
auto	**else**	operator	template
break	**enum**	private	this
case	**extern**	protected	throw
catch	**float**	public	try
char	**for**	**register**	**typedef**
class	friend	**return**	**union**
const	**goto**	**short**	**unsigned**
continue	**if**	**signed**	virtual
default	inline	**sizeof**	**void**
delete	**int**	**static**	**volatile**
do	**long**	**struct**	**while**
Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t
explicit	namespace	typeid	

Identifiers

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letters are distinct.
- A declared keyword cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, ANSI C++ places no limit on its length and, therefore, all the characters in a name are significant.

Constants

Constants refer to fixed values that do not change during the execution of a program.

```
123             // decimal integer
12.34           // floating point integer
037             // octal integer
0X2             // hexadecimal integer
"C++"           // string constant
'A'             // character constant
L'ab'           // wide-character constant
```

The **wchar_t** type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte. Wide-character literals begin with the letter **L.**

C++ also recognizes all the backslash character constants available in C.

Strings

C++ supports two types of string representation - the C-style character string and the string class type introduced with Standard C++.

Basic Data types

Data types in C++ can be classified under various categories *as* shown in Fig. 5.

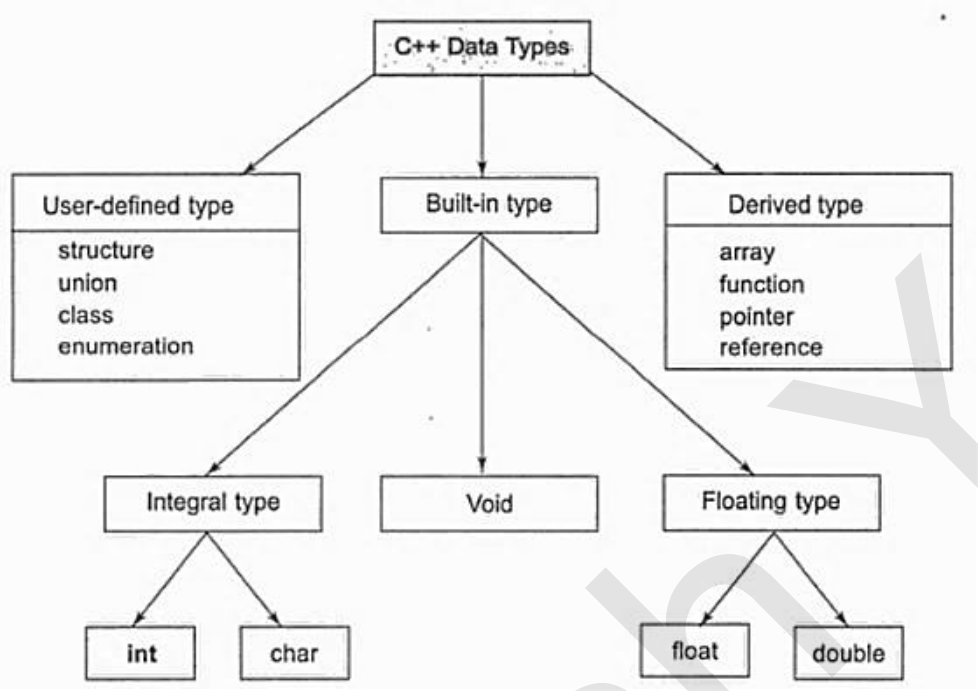


Fig. 5 C++ data types

* Both C and C++ compilers support all the built-in (also known as *basic* or *fundamental)* data types.
* With the exception of **void**, the basic data types may have several ***modifiers*** preceding them to serve the needs of various situations. The modifiers **signed**, **unsigned**, **long**, **and short** may be applied to character and integer basic data types.
* However, the modifier **long** may also be applied to **double**. **Data type representation is machine specific in C++.**

Table 2: size and ranges of C++ basic data types (for 16-bit word machine)

Type	Bytes	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	– 31768 to 32767
short int	2	– 31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	–32768 to 32767
long int	4	–2147483648 to 2147483647
signed long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E–38 to 3.4E+38
double	8	1.7E–308 to 1.7E+308
long double	10	3.4E–4932 to 1.1E+4932

The fallowing explanation of void data type can be understood properly after discussion of pointers in module 4

The type **void** was introduced in ANSI C. Two normal uses of **void** are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void funct1(void);
```

Another interesting use of **void** is in the declaration of generic pointers. Example:

```
void *gp; // gp becomes generic pointer
```

A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example,

```
int *ip;  // int pointer
gp = ip;  // assign int pointer to void pointer
```

are valid statements. But, the statement,

```
*ip = *gp;
```

is illegal. It would not make sense to dereference a pointer to **a void** value.

Assigning any pointer type to a **void** pointer without using a cast is allowed in both C++ and ANSI C. In ANSI C, we can also assign a **void** pointer to a non-void pointer without using a cast to non-void pointer type. This is not allowed in C++. For example,

```
void *ptr1;
char *ptr2;
ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
ptr2 = (char *) ptr1;
```

User-Defined Data Types

Structures

Standalone variables of primitive types are not sufficient enough to handle real world problems. It is often required to group logically related data items together. While arrays are used to group together similar type data elements, structures are used for grouping together elements with dissimilar types.

The **general format** of a structure definition is as follows:

```
struct   name
{
     Variable name1;
     Variable name2;
     . . . . . .
     . . . . . .
};
```

Consider an example of a book, which has several attributes such as title, number of pages, price etc. we can realize a book using structures as shown below:

```
struct   book
{
     char title[25];
     char author[25];
     int pages;
     float price;
};
struct book   book1, book2, book3;
```

here book1, book2 and book3 are declared as variables of the user-defined type book. We can access the member elements by dot(.) operator as

```
book1.pages=300;
book2.price=275.75;
```

Unions

Unions are conceptually similar to structures as they allow us to group together dissimilar type elements inside a single unit.

```
union   book
{
     char title[25];
     char author[25];
     int pages;
     float price;
};
```

But there are significant differences between structures and unions as far as their implementation is concerned.

The size of a structure type is equal to the sum of the sizes of individual member types. However, the size of a union is equal to the size of its largest member element.

Table 3: Differences between structures and unions

Structures	Unions
1.The keyword struct is used to define a structure	1. The keyword union is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused slack bytes.	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
3. Each member within a structure is assigned unique storage area of location.	3. Memory allocated is shared by individual members of union.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.

Class

C++ also permits us to define another user-defined data type known as class which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus or object-oriented programming.

Enumerated Data Type

* An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code**.**
* The enum keyword (from C) automatically enumerates a list of words by assigning them values 0,1,.2. and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the struct statement.

Example:

```
enum shape {circle, square, triangle};
enum colour {red, blue, green, yellow};
enum position {off, on};
```

In C++, the tag names **shape, colour, and position** become new type names. By using these tag names, we can declare new variables.

```
colour background = blue;    // allowed
colour background = 7;       // Error 1n C++
colour background = (colour) 7; //OK
```

However, an enumerated value can be used in place of an int value,

```
int c = red; // valid colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can override the default by explicitly assigning integer values to the enumerators. For example,

```
enum colour {red, blue=4, green=6};
enum colour {red =5, blue, green};
```

C++ also permits the creation of anonymous enums (i.e., enums without tag names).

```
enum {off, on};
```

Here, off is 0 and on is 1. These constants may be referenced in the same manner as regular constants.

```
int switch1 = off;
int switch2 = on;
```

Derived Data Types

Arrays

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

```
char string [3]= "xyz";
```

is valid in ANSI C. It assumes that the programmer intends to leave out the null character (\0) in the definition. But in C++, the size should be one larger than the number of characters in the string.

```
char string[4] = "xyz"; // OK for C++
```

Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++.

Pointers

Pointers are declared and initialized as in C. Examples:

```
int *ip; // int pointer
ip = &x; // address of x assigned to ip
*ip =10; // 10 assigned to x through indirection
```

C++ adds the concept of constant pointer and pointer to a constant.

```
char *const ptr1 ="Yes"; // constant pointer
```

We cannot modify the address that **ptr1** is initialized to.

```
int const *ptr2 = &m; // pointer to a constant
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

```
const char * const cp = "xyz";
```

This statement declares cp as a constant pointer to the string which has been declared a constant. In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed.

Symbolic Constants

There are two ways of creating symbolic constants in C++:

- Using the qualifier **const** and
- Defining a set of integer constants using **enum** keyword.

```
const int size= 10;
char name[size];
```

This would be illegal in C but valid in C++. **const** allows us to create typed constants instead of having to use **#define** to create constants that have no type information.

As with **long** and **short**, if we use the **const** modifier alone, it defaults to int. For example,

```
const size =10;
//means
const int size =10;
```

C++ requires a const to be initialized. ANSI C does not require an initializer; if none is given, it initializes the const to 0.

The scoping of const values differs. A const in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. In ANSI C, const values are global in nature.

They are visible outside the file in which they are declared. However, they can be made local by declaring them as **static** .

To give a const value an external linkage so that it can be referenced from another file. we must explicitly define it as an **extern** in C++. Example:

```
extern const total = 100;
```

Another method of naming integer constants is by enumeration as under;

```
enum {X, Y, Z};
```

This defines X. Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to:

```
const X=0;
const Y=1;
const Z=2;
```

We can also assign values to X, Y, and Z explicitly. For example:

```
enum {X=100, Y=50,Z=200};
```

Declaration of Variables

We know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well.

```
int main()
{
    float x;  //declaration
    float sum = 0;
    for (int i=1;i<5;i++)   //declaration
    {
        cin >> x;
        sum= sum +x;
    }
    float average;     //declaration
    average = sum/(i-1):
    cout << average;
    return 0;
}
```

Dynamic Initialization of Variables

C++ permits initialization of the variables at run time. This is referred to as ***dynamic initialization***, In C++, a variable can be initialized at run time using expressions at the place of declaration.

For example

```
……….
int n = strlen(string);
……….
……….
float area =3.14159 * rad * rad;
……….
```

Dynamic initialization is extensively used in object oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

Reference Variables

C++ introduces a new kind of variable known as the *reference* variable. A reference variable provides an *alias* (alternative name) for a previously defined variable.

A reference variable is created as follows:

```
data_type & reference_name = variable_name;
```

For example, if we make the variable **sum** a reference to the variable **total**, then sum and total can be used interchangeably to represent that variable.

```
float total=100;
float & sum= total;

cout « total;
```

and

```
cout << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Reference variables are used as function arguments, which will be discussed in call by reference method.

Where Variables Are Declared

Variables will be declared in three basic places: inside functions, in the definition of function parameters, and outside of all functions. These are local variables, formal parameters, and global variables.

Local Variables

* Variables that are declared inside a function are called *local variables*. In some C/C++ literature, these variables are referred to as ***automatic*** variables.
* Local variables exist only while the block of code in which they are declared is executing. That is, a local variable is created upon entry into its block and destroyed upon exit.

For example, consider the following two functions:

```
void func1(void)
{
     int x;
     x = 10;
}
void func2(void)
{
     int x;
     x = -199;
}
```

Formal Parameters

* If a function is to use arguments, it must declare variables that will accept the values of the arguments. These variables are called the ***formal parameters*** of the function.
* They behave like any other local variables inside the function. As shown in the following program fragment, *their declarations occur after the function name and inside parentheses:*

```
int fun1( char c)
{
     c='a';
     return 0;
}
```

Global Variables

* Unlike local variables, *global variables* are known throughout the program and may be used by any piece of code. Also, they will hold their value throughout the program's execution.

* We can create global variables by declaring them outside of any function. Any expression may access them, regardless of what block of code that expression is in.

```
#include <stdio.h>
int count;    /* count is global */
void func1(void);
int main(void)
{
     count = 100;
     func1();
     return 0;
}
void func1(void)
{
     int temp;
     temp = count;
     cout <<"count is"<<count; /* will print 100 */
}
```

Storage classes

There are four storage class specifiers supported by C++:

extern
auto
static
register
mutable

These specifiers tell the compiler how to store the subsequent variable. The general form of a declaration that uses one is shown here.

```
storage_specifier type var_name;
```

extern

Because C/C++ allows separate modules of a large program to be separately compiled and linked together, there must be some way of telling all the files about the global variables required by the program. Although C technically allows you to define a global variable more than once, it is not good practice (and may cause problems when linking).

More importantly, in C++, you may define a global variable *only once* and inform all files in program about these variables.

File One

```
int x, y;
char ch;
int main(void)
{
     /* ... */
}
void func1(void)
{
     x = 123;
}
```

File Two

```
extern int x, y;
extern char ch;
void func22(void)
{
     x=y/10;
}
void func23(void)
{
     y=10;
}
```

Automatic

Automatic storage class assigns a variable to its default storage type. *auto* keyword is used to declare automatic variables.

However, if a variable is declared without any keyword inside a function, it is automatic by default. This variable is **visible** only within the function it is declared and its **lifetime** is same as the lifetime of the function as well. Once the execution of function is finished, the variable is destroyed.

Syntax of Automatic Storage Class Declaration

```
datatype var_name1 [= value];
                or
auto datatype var_name1 [= value];
```

Example of Automatic Storage Class

```
auto int x;
float y = 5.67;
```

Static

Static storage class ensures a variable has the **visibility** mode of a local variable but **lifetime** of an external variable. It can be used only within the function where it is declared but destroyed only after the program execution has finished.

When a function is called, the variable defined as static inside the function retains its previous value and operates on it. This is mostly used to save values in a recursive function.

For example,

```
static int x = 101;
static float sum;
```

Register

Register storage assigns a variable's storage in the CPU registers rather than primary memory. It has its lifetime and visibility same as automatic variable.

The purpose of creating register variable is to increase access speed and makes program run faster. If there is no space available in register, these variables are stored in main memory and act similar to variables of automatic storage class. So only those variables which requires fast access should be made register.

For example,

```
register int id;
register char a;
```

Example of Storage Class

```
//C++ program to create automatic, global, static and register
variables.
    #include<iostream>
    using namespace std;
    int g;    //global variable, initially holds 0

    void test_function()
    {
        static int s;    //static variable, initially holds 0
        register int r;    //register variable
        r=5;
        s=s+r*2;
        cout<<"Inside test_function"<<endl;
        cout<<"g = "<< g <<endl;
        cout<<"s = "<< s <<endl;
        cout<<"r = "<< r <<endl;
    }

    int main()
    {
        int a;    //automatic variable
        g=25;
        a=17;
        test_function();
        cout<<"Inside main"<<endl;
        cout<<"a = "<<a<<endl;
        cout<<"g = "<<g<<endl;
        test_function();
        return 0;
    }
```

In the above program, *g* is a global variable, *s* is static, *r* is register and *a* is automatic variable.

We have defined two function, first is ***main()*** and another is ***test_function().***

Since *g* is global variable, it can be used in both function. Variables *r* and *s* are declared inside ***test_function()*** so can only be used inside that function.

However, ***s*** being static isn't destroyed until the program ends. When *test_function()* is called for the first time, *r* is initialized to 5 and the value of *s* is 10 which is calculated from the statement,

```
s=s+r*2;
```

After the termination of *test_function()*, *r* is destroyed but *s* still holds 10. When it is called second time, *r* is created and initialized to 5 again.

Now, the value of *s* becomes 20 since *s* initially held 10. Variable *a* is declared inside *main()* and can only be used inside *main()*.

Output

```
Inside test_function
g = 25
s = 10
r = 5
Inside main
a = 17
g = 25
Inside test_function
g = 25
s = 20
r = 5
```

Mutable

In C++, a class object can be kept constant using keyword *const*. This doesn't allow the data members of the class object to be modified during program execution. But, there are cases when some data members of this constant object must be changed.

For example, during a bank transfer, a money transaction has to be locked such that no information could be changed but even then, its state has to be changed from - ***started*** to ***processing*** to ***completed*****.** In those cases, we can make these variables modifiable using a **mutable** storage class.

Syntax for Mutable Storage Class Declaration

```
mutable datatype var_name1;
```

For example,

```
mutable int x;
mutable char y;
```

Example of Mutable Storage Class

```
// C++ program to create mutable variable.
    #include<iostream>
    using namespace std;

    class test
    {
        mutable int a;
        int b;
        public:
            test(int x,int y)
            {
                a=x;
                b=y;
            }
            void square_a() const
            {
                a=a*a;
            }
            void display() const
            {
                cout<<"a = "<<a<<endl;
                cout<<"b = "<<b<<endl;
            }
    };

    int main()
    {
        const test x(2,3);
        cout<<"Initial value"<<endl;
        x.display();
        x.square_a();
        cout<<"Final value"<<endl;
        x.display();
        return 0;
    }
```

A class *test* is defined in the program. It consists of a mutable data member *a*. A constant object *x* of class test is created and the value of data members are initialized using user-defined constructor.

Since, *b* is a normal data member, its value can't be changed after initialization. However *a* being mutable, its value can be changed which is done by invoking *square_a()* method. *display()* method is used to display the value the data members.

Output

Initial value	Final value
a = 2	a = 4
b = 3	b = 3

Storage Class	Keyword	Lifetime	Visibility	Initial Value	Storage	Purpose
Automatic	auto	Function Block	Local	Garbage	Stack segment	Local variables used by a single function
External	extern	Whole Program	Global	Zero	Data segment	Global variables used throughout the program
Static	static	Whole Program	Local	Zero	Data segment	Local variables retaining their values throughout the program
Register	register	Function Block	Local	Garbage	CPU registers	Variables using CPU for storage purpose
Mutable	mutable	Class	Local	Garbage	Depends on the scope of class	

Extra info (not in syllabus)

Operators

Operators are the symbols which tell the computer to execute certain mathematical or logical operations. A mathematical or logical expression is generally formed with the help of an operator. C ++ programming offers a number of operators which are classified into different categories viz.

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Bitwise operators
6. Special operators

1. Arithmetic Operators

Operator	Action
–	Subtraction, also unary minu
+	Addition
*	Multiplication
/	Division
%	Modulus
– –	Decrement
++	Increment

Note: '%' cannot be used on floating data type.

C programming allows the use of **++** and **–** operators which are increment and decrement operators respectively. Both the increment and decrement operators are unary operators. The increment operator ++ adds 1 to the operand and the decrement operator – subtracts 1 from the operand. The general syntax of these operators are:

Increment Operator: *m++ or ++m*;

Decrement Operator: *m--or --m*;

In the example above, *m++* simply means *m=m+1;* and *m--* simply means *m=m-1;* Increment and decrement operators are mostly used in for and while loops.

++m and *m++* performs the same operation when they form statements independently but they function differently when they are used in right hand side of an expression.

++m is known as prefix operator and *m++* is known as postfix operator. A prefix operator firstly adds 1 to the operand and then the result is assigned to the variable on

the left whereas a postfix operator firstly assigns value to the variable on the left and then increases the operand by 1. Same is in the case of decrement operator.

2.Relational Operators

Relational operators are used when we have to make comparisons. C programming offers 6 relational operators.

Relational Operators

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
= =	Equal
!=	Not equal

3. Logical Operators

Logical operators are used when more than one conditions are to be tested and based on that result, decisions have to be made. C programming offers three logical operators. They are:

Logical Operators

Operator	Action
&&	AND
| |	OR
!	NOT

4. Assignment Operators

Assignment operators are used to assign result of an expression to a variable. '=' is the assignment operator in C. Furthermore, C also allows the use of shorthand assignment operators. Shorthand operators take the form:

```
var op = exp;
```

5. Bitwise Operator

In C programming, bitwise operators are used for testing the bits or shifting them left or right. The bitwise operators available in C are:

Operator	Action
&	AND
|	OR
^	Exclusive OR (XOR)
~	One's complement (NOT)
>>	Shift right
<<	Shift left

Operator precedence and associativity

Operator	Associativity
::	left to right
–> . ( ) [ ] postfix ++ postfix – –	left to right
prefix ++ prefix – – ~ ! unary + unary – unary * unary & (type) sizeof new delete	right to left
–> * *	left to right
* / %	left to right
+ –	left to right
<< >>	left to right
<< = >> =	left to right
= = !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	left to right
= * = / = % = + = =	right to left
<< = >> = & = ^= |=	left to right
, (comma)	

7. Special operators:-

The ? Operator

C/C++ contains a very powerful and convenient operator that replaces certain statements of the if-then-else form. The ternary operator **?** takes the general form

```
Exp1 ? Exp2 : Exp3;
```

where *Exp1*, *Exp2*, and *Exp3* are expressions.

The **?** operator works like this: *Exp1* is evaluated. If it is true, *Exp2* is evaluated and becomes the value of the expression. If *Exp1* is false, *Exp3* is evaluated and its value becomes the value of the expression. For example, in

```
x = 10;
y = x>9 ? 100 : 200;
```

y is assigned the value 100. If **x** had been less than 9, **y** would have received the value 200. The same code written using the **if-else** statement is

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

The & and * Pointer Operators are discussed in module 4

The Compile-Time Operator *sizeof*

sizeof is a unary compile-time operator that returns the length, in bytes, of the variable or parenthesized type-specifier that it precedes. For example, assuming that integers are 4 bytes and doubles are 8 bytes,

```
double f;
printf("%d", sizeof (f));
printf("%d", sizeof(int));
```

The Comma Operator

The comma operator strings together several expressions. The left side of the comma operator is always evaluated as **void**. This means that the expression on the right side becomes the value of the total comma-separated expression. For example,

```
x = (y=3, y+1);
```

first assigns **y** the value 3 and then assigns **x** the value 4. The parentheses are necessary because the comma operator has a lower precedence than the assignment operator.

The Dot (.) and Arrow (–>) Operators

In C, the **.** (dot) and the **–>**(arrow) operators access individual elements of structures and unions. In C++, the dot and arrow operators are also used to access the members of a class.

For example,

```
struct employee
{
     char name[80];
     int age;
     float wage;
} emp;
struct employee *p = &emp; /* address of emp into p */
```

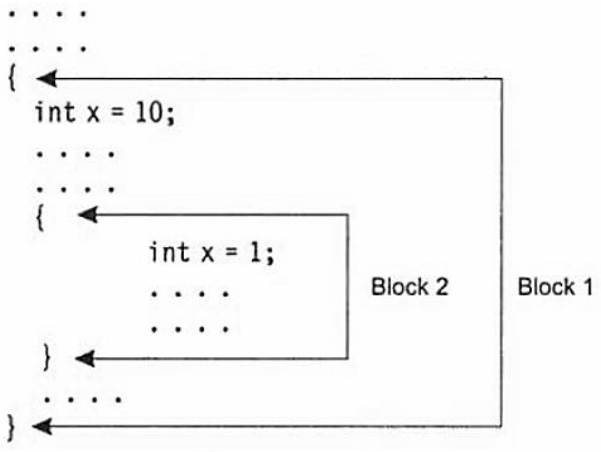
you would write the following code to assign the value 123.23 to the **wage** member of structure variable **emp**:

```
emp.wage = 123.23;
```

However, the same assignment using a pointer to **emp** would be

```
p->wage = 123.23;
```

Scope Resolution Operator



In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the ***scope resolution operator.*** This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

```
#include <iostream>
using namespace std;
int m = 10; // global m
int main()
{
     int m = 20; // m redeclared, local to main
     {
          int k = m;
          int m = 30; // m declared again
             // local to inner block
          cout << "we are in inner block \n";
          cout << "k =" << k << "\n";
          cout << "m =" << m << "\n";
          cout << "::m =" << ::m << "\n";
     }
     cout << "\n We are in outer block \n";
     cout << "m = 11 "<< m << "\n";
     cout << "::m =" << ::m << "\n";
     return 0;
}
```

Output

We are in inner block
k = 20
m = 30
: : m = 10
We are in outer block
m = 20
: : m = 10

Memory Management Operators

* C uses **`malloc()`** and **`calloc()`** functions to allocate memory dynamically at run time. Similarly, it uses the function **`free()`** to free dynamically allocated memory.
* Although C++ supports these functions, it also defines two unary operators **`new`** and **`delete`** that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as ***free store*** operators.

The **new** operator offers the following advantages over the function **`malloc().`**

1. It automatically computes the size of the data object. We need not use the operator **`sizeof.`**

2. It automatically returns the correct pointer type, so that there is no need to use a type cast.

3. It is possible to initialize the object while creating the memory space.

4. Like any other operator, **`new`** and **`delete`** can be overloaded.

* An object can be created by using **new,** and destroyed by using **delete,** as and when required.
* A data object created inside a block with **new,** will remain in existence until it is explicitly destroyed by using **delete.** Thus, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The **new** operator can be used to create objects of any type. It takes the following general form:

```
pointer- variable = new data- type;
```

The **new** operator allocates sufficient memory to hold a data object of type ***data-type*** and returns the address of the object. The *data-type* may be any valid data type. The ***pointer-variable*** holds the address of the memory space allocated.

```
p = new int;
q = new float;
```

```
int *p = new int;
float *q = new float ;
```

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created **int** object and 7.5 to the **float** object.

We can also initialize the allocated memory using the **new** operator. This is done as follows:

```
pointer-variable = new data-type(value);
```

```
int *p = new int(25);
float *q = new float(7.5);
```

similarly, memory for array data type can be allocated as

```
pointer-variable = new data- type[size];
```

```
int *p = new int[10];
```

creates a memory space for an array of 10 integers. **p[0]** will refer to the first element, **p[1]** to the second element, and so on.

When creating multi-dimensional arrays with **new,** all the array sizes must be supplied.

```
array_ptr = new int[3][5][4];    // legal
array_ptr = new int[m][5][4];  // legal
array_ptr = new int[3][5][ ];  // illegal
array_ptr = new int[ ][5][4];   // illegal
```

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
delete pointer-variable;
```

```
delete p;
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of **delete:**

```
delete [size] pointer-variable;
```

Recent versions of C++ do not require the size to be specified. For example,

```
delete [ ] p;
```

will delete the entire array pointed to by **p.**

What happens if sufficient memory is not available for allocation? In such cases, like **malloc(), new** returns a null pointer. Therefore, it may be a good idea to check for the pointer produced by **new** before using it. It is done as follows:

```
..........
..........
p = new int;
if(!p)
{
     cout << "allocation failed \n";
}
..........
..........
```

Member Dereferencing Operators

C++ permits us to define a class containing various types of data and functions as members, C++ also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer-to-member operators.

::*	To declare a Pointer to a member of a class
*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw.**

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n". For example,

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

If we assume the values of the variables as 2597, 14, and 175 respectively, the output will appear as follows:

```
m = 2597
n = 14
p = 175
```

It should rather appear as under:

```
m =  2597
n =    14
p =   175
```

Here, the numbers are *right-justified.* This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The setw manipulator does this job. It is used as follows:

```
cout << setw(5) <<sum<< endl;
```

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

		3	4	5

```
//Use of manipulators
#include <iostream>
#include <iomanip> // for setw
using namespace std;
int main()
{
     long pop1=2425785, pop2=47, pop3=9761;
     cout << setw(8) <<"LOCATION"<< setw(12)<<"POPULATION"<< endl
          << setw(8) <<"Portcity"<< setw(12) << pop1 << endl
          << setw(8) <<"Hightown"<< setw(12) << pop2 << endl
          << setw(8) <<"Lowville"<< setw(12) << pop3 << endl;
     return 0;
}
```

Type Cast Operator

C++ permits explicit type conversion or variables or expressions using the type cast operator.

Traditional C casts are augmented in C++ by a function call notation as a syntactic alternative. The following two versions are equivalent:

```
(type-name) expression // C notation
type-name (expression) // C++ notation
Examples:
average = sum/(float)i; // C notation
average = sum/float(i); // C++ notation
```

A type-name behaves as if it is a function for converting values to a designated type. The function call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example,

```
p = int* (q); // illegal
```

In such cases we must use C type notation.

```
p = (int*) q;
```

Alternatively, we can use **typedef** to create an identifier of the required type and use it in the functional notation.

```
typedef int* int_pt;
p = int_pt(q);
```

Expressions and their types

- Constant Expressions
- Integral Expressions
- Float Expressions
- Pointer Expressions
- Relational expressions
- Logical Expressions
- Bitwise Expressions

Constant Expressions

Constant Expressions consists of only constant values

```
15
12+1/2.0
'x'
```

Integral Expressions

Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

```
m
m = n - 5
m = 'x'
5 + int(2.0)
```

where m and n are integer variables.

Float Expressions

Float Expressions are those which, after all conversions, produce floating-point results.

Examples:

```
x + y
x * y / 10
5 + float(10)
10.75
```

where x and y are floating point variables.

Pointer Expressions

Pointer Expressions produce address values. Examples:

```
&m
ptr
ptr + 1
"xyt"
```

where m is a variable and ptr is a pointer.

Relational Expressions

Relational Expressions yield results of type bool which takes a value true or false. Examples:

```
x <= y
a+b ==c+d
m+n > 100
```

Relational expressions are also known as Boolean expressions.

Logical Expressions

Logical Expressions combine two or more relational expressions and produces bool type results. Examples:

```
a>b  &&  x== 10
x==10 || y==5
```

Bitwise Expressions

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples:

```
x << 3 // Shift three bit position to left
y >> 1 // Shift one bit position to right
```

Shift operators are often used for multiplication and division by powers of two.

Special Assignment Expressions

Chained Assignment

```
x=(y=10);
or
x=y= 10;
```

First 10 is assigned to **y** and then to **x**.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a=b = 12.34; // is illegal.
This may be written as
float a=12.34, b=12.34; // correct
```

Embedded Assignment

```
x =(y = 50) + 10;
```

Here, the value 50 is assigned to y and then the result 50+ 10 = 60 is assigned to x. This statement is identical to

```
y =50;
x = y + 10;
```

Compound Assignment

Like C, C++ supports a *compound assignment operator* which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x = x + 10;
```

may be written as

```
x += 10;
```

The operator += is known as ***compound assignment operator*** or ***short-hand assignment operator.*** The general form of the compound assignment operator is:

```
variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that

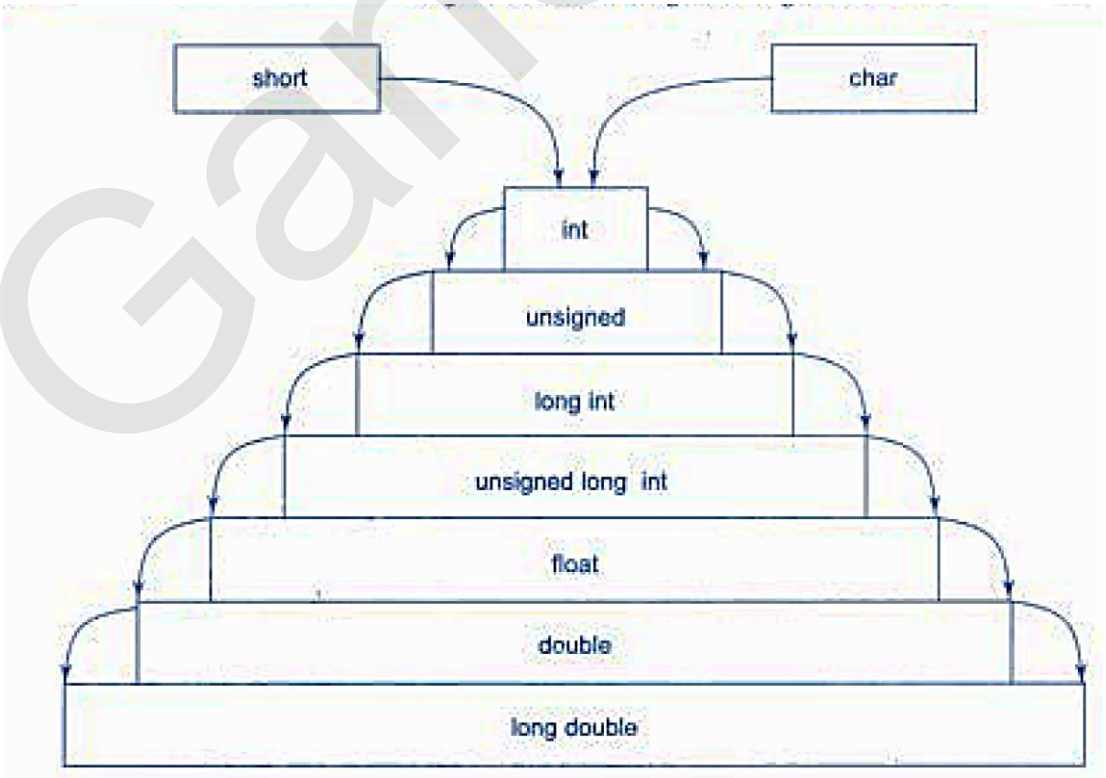
```
variable1 = variable1 op variable2;
```

Implicit Conversions

We can mix data types in expressions. For example,

```
m = 5 + 2.75;
```

is a valid statement. Wherever data types are mixed in an expression. C++ performs the conversions automatically. This process is known as ***implicit or automatic conversion.***



When the compiler encounters an expression, it divides the expressions into sub expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ the compiler converts one of them to match with the other, using the rule that the ***"smaller" type is converted to the "wider" type***.

Results of Mixed-mode Operations

LHO \ RHO	char	short	int	long	float	double	long double
char	int	int	int	long	float	double	long double
short	int	int	int	long	float	double	long double
int	int	int	int	long	float	double	long double
long	long	long	long	long	float	double	long double
float	float	float	float	float	float	double	long double
double	double	double	double	double	double	double	long double
long double	long double	long double	long double	long double	long double	long double	long double

RHO – Right-hand operand *LHO – Left-hand operand*

For example, if one of the operand is an **int** and the other is a **float**, the **int** is converted into a **float** because a **float** is wider than an **int.** The "waterfall" model shown in above figure illustrates this rule.

Whenever a **char** or **short int** appears in an expression, it is converted to an **int.** This is called ***integral widening conversion.*** The implicit conversion is applied only after completing all integral widening conversions.

Operator Overloading

Overloading means assigning different meanings to an operation, depending on the context.

For example, the operator ***** when applied to a pointer variable gives the value pointed by the pointer. But it is also commonly used for multiplying two numbers. The number and type of operands decide the nature of operation to follow.

The input/output operators << and >>are good examples of operator overloading. Although the built-in definition of the << operator is for shifting of bits, it is also used for displaying the values of various data types. This has been made possible by the header file iostream where a number of overloading definitions for << are included.

Thus, the statement

```
cout<<75.86;
```

invokes the definition for displaying a **double** type value, and

```
cout<<"well done";
```

invokes the definition for displaying a **char** value. However, none of these definitions in iostream affect the built-in meaning of the operator.

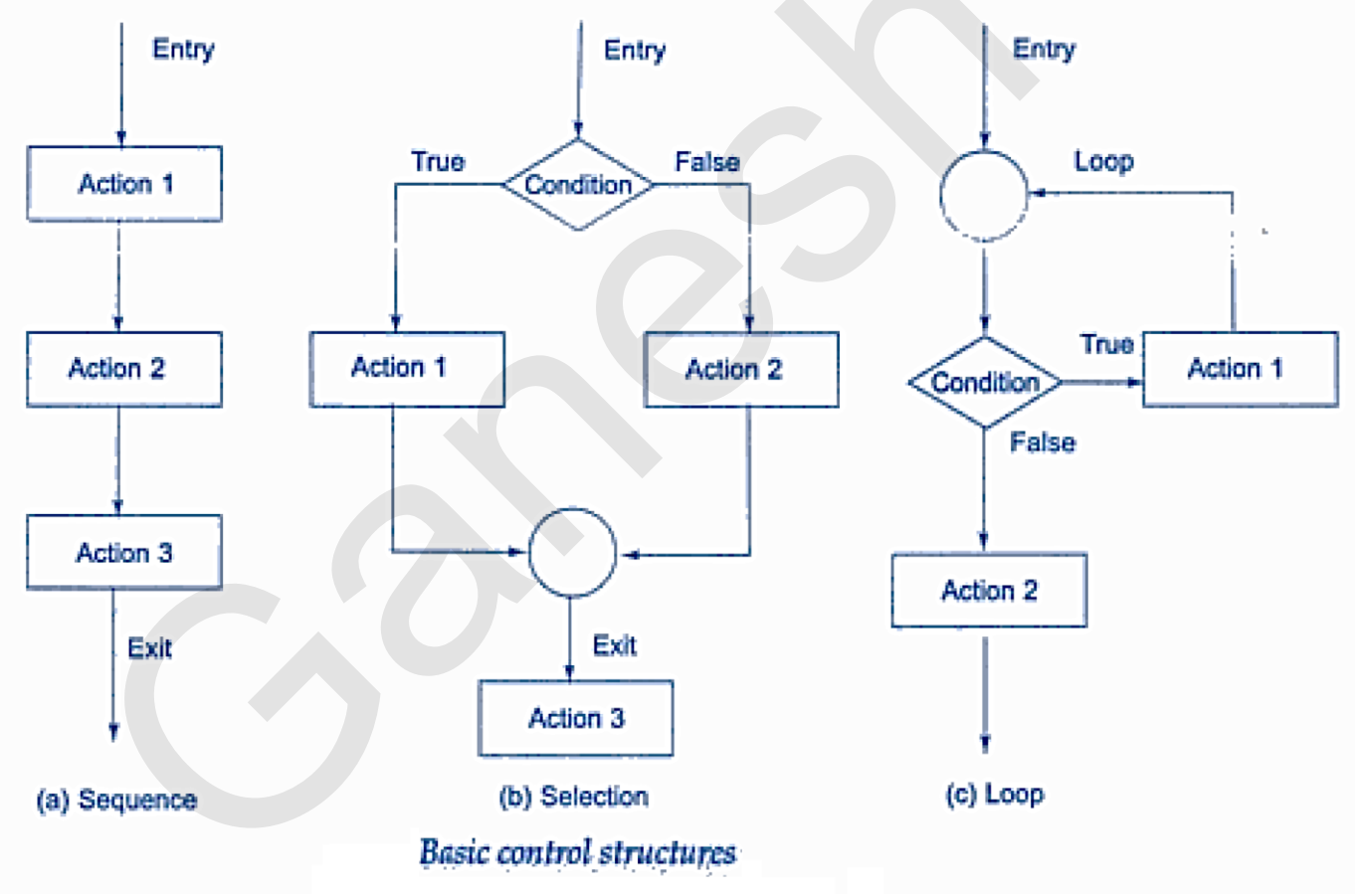
Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (`.and .*`), conditional operator (`?:`), scope resolution operator (`::`) and the size operator (`sizeof`).

Control Structures

One method of achieving the objective of an accurate, error resistant and maintainable code is to use one or any combination of the following three control structures:

1. **Sequence structure (straight line)**
2. **Selection structure (branching)**
3. **Loop structure (iteration or repetition)**

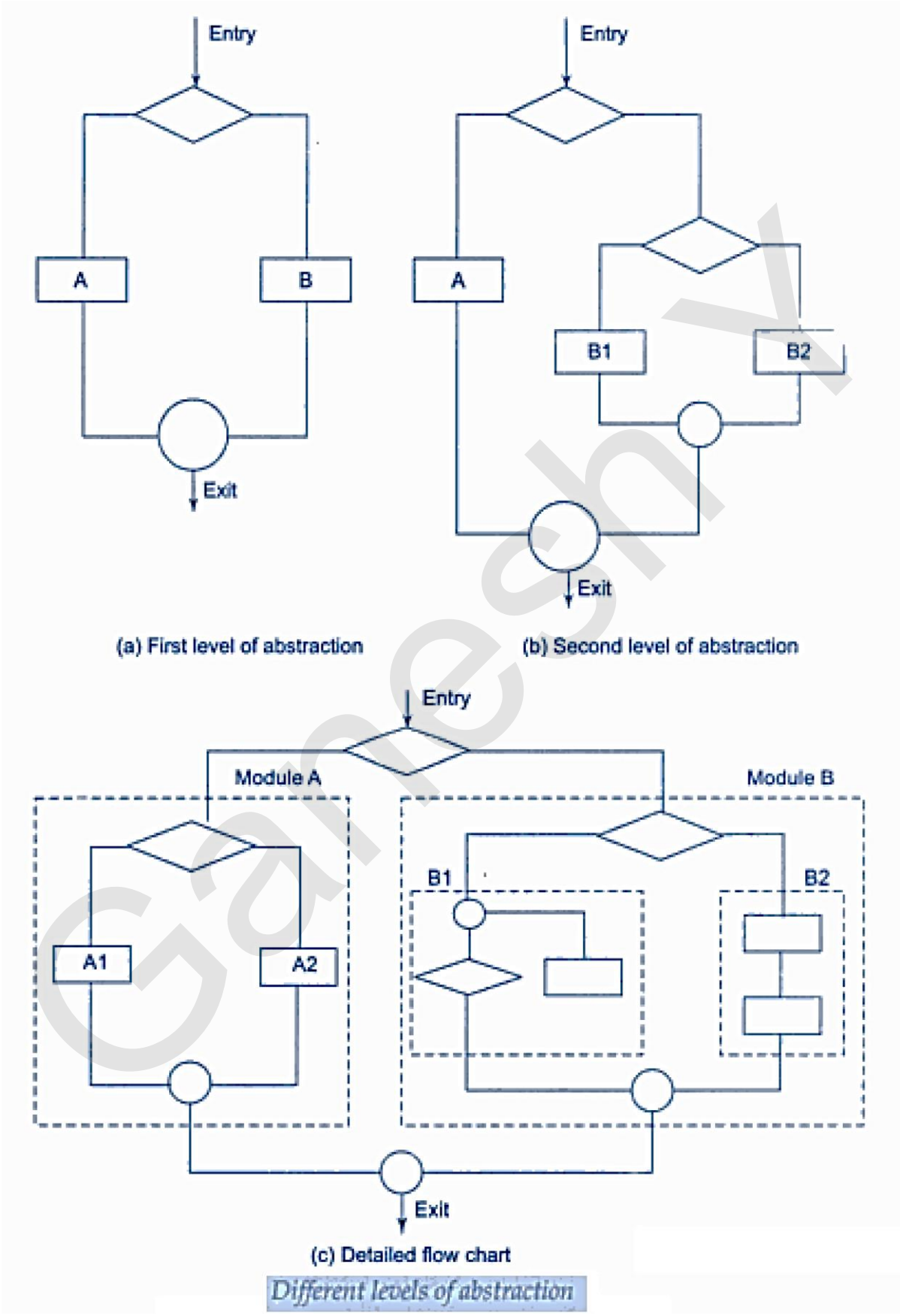
Figure below shows how these structures are implemented using *one-entry, one-exit* concept, a popular approach used in modular programming.



Basic control structures

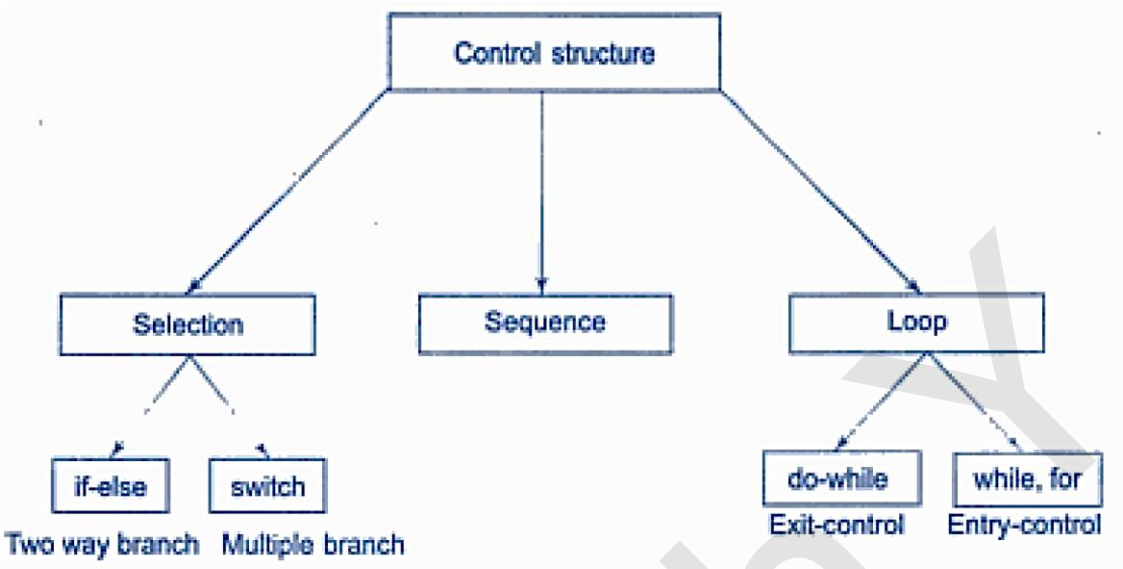
It i s important to understand that all program processing can be coded by using only these three logic structures. The approach of using one or more of these basic control constructs in programming is known as ***structured programming,*** an important technique in software engineering.

Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown in following Figs (a), (b) and (c).



Different levels of abstraction

Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in following Fig. This shows that *C++ combines the power of structured programming with the object-oriented paradigm.*



C++ statements to implement in two forms

The if statement

The if statement is implemented in two forms:

- Simple **if** statement
- **If.....else** statement

Examples:

Form 1

```
if(expression is true)
{
     action1;
}
action2;
action3;
```

Form 2

```
if (expression is
true)
{
     action1;
}
else
{
     action2;
}
action3;
```

The switch statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
switch(expression)
{
     case 1:
     {
          action1;
     }
     case 2:
     {
          action2;
     }
     case 3:
     {
          action3;
     }
     default:
     {
          action4;
     }
}
action5;
```

The do-while statement

The do-while is an *exit-controlled* loop. Based on a condition, the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
     action1;
}
while(condition is true);
act1on2;
```

The while statement

This is also a loop structure, but is an *entry-controlled* one. The syntax is as follows:

```
while(condition is true)
{
     action1;
}
action2;
```

The for statement

The for is an *entry-controlled* loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for(initial value; test; increment)
{
     action1;
}
action2;
```

Exercise Questions and Solutions

2.1 *State whether the following statements are TRUE or FALSE.*

(a) *Since C is a subset of C++, all C programs will run under C++ compilers.*

(b) *In C++, a function contained within a class is called a member function.*

(c) *Looking at one or two lines of code, we can easily recognize whether a program is written in C or C++.*

(d) *In C++, it is very easy to add new features to the existing structure of an object.*

(e) *The concept of using one operator for different purposes is known as oerator overloading.*

(f) *The output function printf() cannot be used in C++ programs.*

2.2 *Why do we need the preprocessor directive #include <iostream> ?*

2.3 *How does a main() function in C++ differ from main() in C?*

2.4 *What do you think is the main advantage of the comment / / in C++ as compared to the old C type comment?*

2.5 *Describe the major parts of a C++ program.*

Debugging Exercises

2.1 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
        int i = 0;
        i = i + 1;
        cout << i << " ";
        /*comment\*//i = i + 1;
        cout << i;
}
```

2.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
        short i=2500, j=3000;
        cout >> "i + j = " >> -(i+j);
}
```

2.3 What will happen when you run the following program?

```
#include <iostream.h>
void main()
{
        int i=10, j=5;
        int modResult=0;
        int divResult=0;

        modResult = i%j;
        cout << modResult << " ";

        divResult = i/modResult;
        cout << divResult;
}
```

2.4 Find errors, if any, in the following C++ statements.

(a) cout << "x=" x;
(b) m = 5; // n = 10; // s = m + n;
(c) cin >>x; >>y;
(d) cout << \n "Name:" << name;
(e) cout <<"Enter value:"; cin >> x;
(f) /*Addition*/ z = x + y;

Programming Exercises

2.1 *Write a program to display the following output using a single cout statement.*

Maths = 90
Physics = 77
Chemistry = 69

2.2 *Write a program to read two numbers from the keyboard and display the larger value on the screen.*

2.3 *Write a program to input an integer value from keyboard and display on screen "WELL DONE" that many times.*

2.4 *Write a program to read the values of a, b and c and display the value of x, where*

$$x = a / b - c$$

Test your program for the following values:

(a) *a = 250, b = 85, c = 25*
(b) *a = 300, b = 70, c = 70*

2.5 *Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius.*

2.6 *Redo Exercise 2.5 using a class called* **temp** *and member functions.*

2.1: State whether the following statements are TRUE or FALSE.
(a) Since C is a subset of C++, all C peograms will run under C++ compilers.
(b) In C++, a function contained within a class is called a member function.
(c) Looking at one or two lines of code, we can easily recognize whether a program is written in C or C++.
(d) In C++, it is very easy to add new features to the existing structure of an object.
(e) The concept of using one operator for different purposes is known as aerator overloading. 10
The output function printfl) cannot be used in C++ programs.

Ans:
a> FALSE
b> TRUE
c> FALSE
*** most lines of codes are the same in C & C++
d> TRUE
e> TRUE
f> FALSE

2.2: Why do we need the preprocessor directive #include<iostream>?

Ans: '#include<iostream>' directive causes the preprocessor to add-the contents of iostream file to the program.
2.3: How does a main() function in C++ differ from main{} in C?

Ans: In C main () by default returns the void type but in C++ it returns integer by default.
2.4: What do you think is the main advantage of the comment / / in C++ as compared to the old C type comment?

Ans: '//' is more easy and time-saving than '/* */'

2.5:Describe the major parts of a C++ program.

Ans: Major parts of a C++ program :
1. Include files
2. Class declaration
3. Member function definitions
4. Main function program

Debugging Exercises

2.1: **Identify the error in the following program.**

```
#include<iostream.h>
void main()
{
   int i = 0;
   i = i + 1;
   cout « i « " ";
   /*comment \*//i = i + 1;
   cout << i;
}
```

Ans: Syntax error→/* comment*//i=i+1;

2.2: **Identify the error in the following program.**

```
#include<iostream.h>
void main()
{
    short i=2500, j=3000;
    cour>> "i+j=">> -(i+j);
}
```

Ans: cout >> "i+j=">> Illegal structure operation.→-(i + j);

2.3: **What will happen when you run the following program?**

```
#include<iostream.h>
void main()
{
    int i=10, j=5;
    int modResult=0;
    int divResult=0;
    modResult = i%j;
    cout<<modResult<<" ";
    divResult = i/modResult;
    cout<<divResult;
}
```

Ans: floating point Error or divide by zero→divResult = i/modResult;

Note: If this kind of Error exist in a program, the program will successfully compile but it will show Run Error.

2.4: **Find errors, if any, in the following C++ statements.**
(a) cout.<<"x=" x; (b) m = 5; // n = 10; // = m + n: (c) cin >>x; >>y;
(d) cout <<\h 'Name:" <<name;
(e) cout <<"Enter value:"; cin >> x:
(f) /*Addition*/ z = x + y;

Ans:

	Error	Correction
A	Statement missing	cout<<"x="<<x;
B	No error	
C	Expression-syntax-error	cin>>x>>y;
D	Illigal character '\'. Statement missing	cout<<"\n Name"<<name;
E	No error	
F	No error	

Programming Exercises

2.1: **Write a program to display the following output using a single cout statement**

Maths = 90
Physics = 77
Chemistry = 69

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  int main()
4  {
5
6           char *sub[]={"Maths","Physics","Chemestry"};
7           int mark[]={90,77,69};
8           for(int i=0;i<3;i++)
9            {
10cout<<setw(10)<<sub[i]<<setw(3)<<"="<<setw(4)<<mark[i]<<endl;
11           }
12    return 0;
13}
```

output

```
Maths = 90
Physics = 77
Chemistry = 69
```

2.2: **Write a program to read two numbers from the keyboard and display the larger value on the screen.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3
4  int main()
5  {
6     float a,b;
7        cout<<" Enter two values  :"<<endl;
8     cin>>a>>b;
9     if(a>b)
10       cout<<" larger value = "<<a<<endl;
11    else
12       cout<<" larger value = "<<b<<endl;
13    return 0;
14}
```

output

```
Enter two values : 10 20
larger value = 20
```

2.3: **Write a program to input an integer from the keyboard and display on the screen "WELL DONE" that many times.**

Solution:

```
1  Solution:
2  #include<iostream.h>
3  #include<iomanip.h>
4  int main()
5  {
6     int n;
7     char *str;
8     str="WELL DONE";
9     cout<<" Enter an integer value ";
10    cin>>n;
11    for(int i=0;i<n;i++)
12    {
13       cout<<str<<endl;
14    }
15    return 0;
16}
```

output

```
Enter an integer value 5
WELL DONE
WELL DONE
WELL DONE
WELL DONE
WELL DONE
```

2.4: **Write a program to read the values a, b and c and display x, where x = a / b –c.**

Test the program for the following values:
(a) a = 250, b = 85, c = 25
(b) a = 300, b = 70, c = 70

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  int main()
4  {
5      float a,b,c,x;
6      cout<<" Enter the value of a,b, &c :"<<endl;
7      cin>>a>>b>>c;
8      if((b-c)!=0)
9      {
10         x=a/(b-c);
11         cout<<" x=a/(b-c) = "<<x<<endl;
12     }
13     else
14     {
15        cout<<"  x= infinity "<<endl;
16     }
17     return 0;
18}
```

During First Run:

output

Enter the value of a,b, &c : 250 85 25
x=a/(b-c) = 4.166667

During Second Run:

output

Enter the value of a,b, &c : 300 70 70
x= infinity

2.5: **Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3
4  int main()
5  {
6      float f,theta;
7      cout<<" Enter  the temperature in Feranhite    scale  : ";
8      cin>>f;
9      theta=((f-32)/9)*5;
10     cout<<" Temperature in Celsius =  "<<theta<<endl;
11     return 0;
12}
```

output

Enter the temperature in Feranhite scale : 105
Temperature in Celsius = 40.555557

2.6: **Redo Exercise 2.5 using a class called temp and member functions.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3
4  class temp
5  {
6      float f,theta;
7  public:
8      float conversion(float f);
9  };
10
11float temp::conversion(float f)
12{
13    theta=((f-32)/9)*5;
14    return theta;
15}
16int main()
17{
18    temp t;
19    float f;
20    cout<<" Enter temperature in Farenheite scale :"<<endl;
21    cin>>f;
22    cout<<" Temperature in Celsius scale = "<<t.conversion(f)<<endl;
23    return 0;
24}
```

output

```
Enter the temperature in Feranhite scale : 112
Temperature in Celsius = 44.444443
```

3.2 *An **unsigned int** can be twice as large as the **signed int**. Explain how?*

3.3 *Why does C++ have type modifiers?*

3.4 *What are the applications of **void** data type in C++?*

3.5 *Can we assign a **void** pointer to an **int** type pointer? If not, why? How can we achieve this?*

3.6 *Describe, with examples, the uses of enumeration data types.*

3.7 *Describe the differences in the implementation of **enum** data type in ANSI C and C++.*

3.8 *Why is an array called a derived data type?*

3.9 *The size of a **char** array that is declared to store a string should be one larger than the number of characters in the string. Why?*

3.10 *The **const** was taken from C++ and incorporated in ANSI C, although quite differently. Explain.*

3.11 *How does a constant defined by **const** differ from the constant defined by the preprocessor statement **#define**?*

3.12 *In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?*

3.13 *What do you mean by dynamic initialization of a variable? Give an example.*

3.14 *What is a reference variable? What is its major use?*

3.15 *List at least four new operators added by C++ which aid OOP.*

3.16 *What is the application of the scope resolution operator :: in C++?*

3.17 *What are the advantages of using **new** operator as compared to the function **malloc()**?*

3.18 *Illustrate with an example, how the **setw** manipulator works.*

3.19 *How do the following statements differ?*

(a) *char * const p;*

(b) *char const *p;*

Debugging Exercises

3.1 What will happen when you execute the following code?

```
#include <iostream.h>
void main()
{
        int i=0;
        i=400*400/400;
        cout << i;
}
```

3.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
```

```
{
        int num[]={1,2,3,4,5,6};
        num[1]==[1]num  ?    cout<<"Success"    :     cout<<"Error";
}
```

3.3 Identify the errors in the following program.

```
#include <iostream.h>
void main()
{
  int i=5;
  while(i)
  {
          switch(i)
          {
          default:
          case 4:
          case 5:

          break;

          case 1:
          continue;

          case 2:
          case 3:
          break;

          }
          i-;
  }
}
```

3.4 Identify the error in the following program.

```
#include <iostream.h>
#define pi 3.14
int squareArea(int &);
int circleArea(int &);

void main()
{
      int a=10;
      cout << squareArea(a) << " ";
```

```
        cout << circleArea(a) << " ";
        cout << a << endl;
}

int squareArea(int &a)
{
        return a *== a;
}

int circleArea(int &r)
{
        return r = pi * r * r;
}
```

3.5 Identify the error in the following program.

```
#include <iostream.h>
#include <malloc.h>

char* allocateMemory();

void main()
{
        char* str;
        str = allocateMemory();
        cout << str;
        delete str;
        str = "      ";
        cout << str;
}

char* allocateMemory()
{
        str = "Memory allocation test, ";
        return str;
}
```

3.6 Find errors, if any, in the following C++ statements.

(a) long float x;
(b) char *cp = vp; // vp is a void pointer
(c) int code = three; // three is an enumerator
(d) int *p = new; // allocate memory with new
(e) enum (green, yellow, red);
(f) int const *p = total;
(g) const int array_size;
(h) for (i=1; int i<10; i++) cout << i << "\n";

(i) int & number = 100;
(j) float *p = new int [10];
(k) int public = 1000;
(l) char name[3] = "USA";

Programming Exercises

3.1 *Write a function using reference variables as arguments to swap the values of a pair of integers.*

3.2 *Write a function that creates a vector of user-given size **M** using **new** operator.*

3.3 *Write a program to print the following output using **for** loops.*

```
1
22
333
4444
55555
.....
```

3.4 *Write a program to evaluate the following investment equation*

$V = P(1 + r)^n$

and print the tables which would give the value of V for various combination of the following values of P, r and n:

P: 1000, 2000, 3000, ..., 10,000

r: 0.10, 0.11, 0.12, ..., 0.20

n: 1, 2, 3, ..., 10

(Hint: P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as

$V = P(1 + r)$

$P = V$

In other words, the value of money at the end of the first year becomes the principal amount for the next year, and so on.

3.5 *An election is contested by five candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and **count** the votes cast for each candidate using an array variable count. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot', and the program should also count the number of spoilt ballots.*

3.6 *A cricket team has the following table of batting figures for a series of test matches:*

Player's name	Runs	Innings	Times not out
Sachin	8430	230	18
Saurav	4200	130	9
Rahul	3350	105	11
.	.	.	.
.	.	.	.

Write a program to read the figures set out in the above form, to calculate the batting averages and to print out the complete table including the averages.

3.7 *Write programs to evaluate the following functions to 0.0001% accuracy.*

(a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\cdots$

(b) $SUM = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \cdots\cdots$

(c) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots\cdots$

3.8 *Write a program to print a table of values of the function*
$y = e^{-x}$
for x varying from 0 to 10 in steps of 0.1. The table should appear as follows.

TABLE FOR Y = EXP [–X]

X	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.0									
1.0									
.									
.									
.									
9.0									

3.9 *Write a program to calculate the variance and standard deviation of N numbers.*

$$\text{Variance} = \frac{1}{N}\sum_{i=1}^{N}(x_1 - \bar{x})^2$$

$$\text{Standard Deviation} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

where $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_1$

3.10 *An electricity board charges the following rates to domestic users to discourage large consumption of energy:*

For the first 100 units - 60P per unit
For next 200 units - 80P per unit
Beyond 300 units - 90P per unit

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added.

Write a program to read the names of users and number of units consumed and print out the charges with names.

Ans:
In case of unsigned int the range of the input value is : 0 to $2^m - 1$. [where m is no. of bit]
In case of signed int the range of the input value is : -2^{m-1} to + $(2^{m-1} - 1)$

$$\frac{\text{maximum value for unsigned int}}{\text{value for signed int}} = \frac{2^m - 1}{\frac{(2^{m-1}-1) + 2^{m-1}}{2}}$$

$$= \frac{2^m - 1}{\frac{2.2^{m-1}-1}{2}}$$

$$= \frac{2^m - 1}{\frac{2^m-1}{2}}$$

$$= 2$$

So, maximum value for unsigned int can be twice as large as the signed int.
* Here the absolute value of lower value -2^{m-1} for signed int must be considered for finding average value of signed int.

3.3: **Why does C++ have type modifiers?**

Ans: To serve the needs of various situation.

3.4: **What are the applications of void data type in C++?**

Ans: Two normal uses of void are
(1) to specify the return type of a function when it is not returning any value.
(2) To indicate any empty argument list to a function.

Example : void function (void)

Another interesting use of void is in the declaration of generic pointers.

Example :
void *gp; //gp is generic pointer.
A pointer value of any basic data type can be assigned to a generic pointer
int * ip;
gp = ip; // valid.

3.5: **Can we assign a void pointer to an int type pointer? If not, why? Now can we achieve this?**

Ans: We cannot assign a void pointer to an int type pointer directly. Because to assign a pointer to another pointer data type must be matched. We can achieve this using casting.

Example :
void * gp;
int *ip;
ip = (int *) gp
/br<>

3.6: **Describe, with examples, the uses of enumeration data types.**

Ans: An enumerated data type is a user-defined type. It provides a way for attaching names to numbers in ANSIC

Example :
enum kuet (EEE, CSE, ECE, CE, ME, IEM);

The enum keyword automatically enumerates
EEE to 0
CSE to 1
ECE to 2
CE to 3
ME to 4
IEM to 5

In C++ each enumerated data type retains its own separate type.

3.7: **Describe the differences in the implementation of enum data type in ANSI C and C++.**

Ans: Consider the following example :
enum kuet (EEE, CSE, ECE, CE, ME, IEM);

Here, kuet is tag name.

In C++ tag name become new type name. We can declare a new variables Example:

kuet student;
ANSI C defines the types of enum to be int.

In C int value can be automatically converted to on enum value. But in C++ this is not permitted.

Example:
student cgp = 3.01 // Error in C++
// OK in C.

student cgp = (student) 3.01 //OK in C++

3.8: **Why is an army called a derived data type?**

Ans: Derived data types are the data types which are derived from the fundamental data types. Arrays refer to a list of finite number of same data types. The data can be accessed by an index number from o to n. Hence an array is derived from the basic date type, so array is called derived data type.

3.9: **The size of a char array that is declared to store a string should be one larger than the number of characters in the string. Why?**

Ans: An additional null character must assign at the end of the string that's why the size of char array that is declared to store a string should be one larger than the number of characters in the string.

3.10: **The const was taken from C++ and incorporated in ANSI C, although quite differently. Explain.**

Ans: In both C and C++, any value declared as const cannot be modified by the program in any way. However there are some differences in implementation. In C++ we can use const in a constant expression, such as const int size = 10; char name [size];
This would be illegal in C. If we use const modifier alone, it defaults to int. For example, const size = 10; means const int size = 10;
C++ requires const to be initialized. ANSI C does not require an initialization if none is given, it initializes the const to o. In C++ a const is local, it can be made as global defining it as external. In C const is global in nature , it can be made as local declaring it as static.

3.11: **How does a constant defined by cowl differ from the constant defined by the preprocessor statement %define?**

Ans: Consider a example : # define PI 3.14159

The preprocessor directive # define appearing at the beginning of your program specifies that the identifier PI will be replace by the text 3.14159 throughout the program.

The keyword const (for constant) precedes the data type of a variable specifies that the value of a variable will not be changed throughout the program.

In short, const allows us to create typed constants instead of having to use # define to create constants that have no type information.

3.12: **In C++. a variable can be declared anywhere in the scope. What is the significance of this feature?**

Ans: It is very easy to understand the reason of which the variable is declared.

3.13: **What do you mean by dynamic initialization of a variable? Give an example.**

Ans: When initialization is done at the time of declaration then it is know as dynamic initialization of variable

Example :
float area = 3.14159*rad * rad;

3.14: **What is a reference variable? What is its major use?**

Ans: A reference variable provides an alias (alternative name) for a previously defined variable. A major application of reference variables is in passing arguments to functions.

3.15: **List at least four new operators added by C++ which aid OOP.**

Ans:
New opperators added by C++ are :
1. Scope resolution operator ::
2. Memory release operator delete   delete
3. Memory allocation operator   new
4. Field width operator   setw
5. Line feed operator   endl

3.16: **What is the application of the scope resolution operator :: in C++?**

Ans: A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs.

3.17: **What are the advantages of using new operator as compared to the junction ntallocOr**

Ans: Advantages of new operator over malloc ():
1. It automatically computes the size of the data object. We need not use the operator size of.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, new and delete can be overloaded.

3.18: **Illustrate with an example, how the seize manipulator works.**

Ans:
setw manipulator specifies the number of columns to print. The number of columns is equal the value of argument of setw () function.

For example :

setw (10) specifies 10 columns and print the massage at right justified.

cout << set (10) << "1234"; will print

1 2 3 4

If argument is negative massage will be printed at left justified.

cout <<setw(-10)<< "1234"; will print

1 2 3 4

3.19: **How do the following statements differ?**
(a) char *const p;
(b) char canal *p;

Ans:
(a) Char * const P; means constant pointer.
(b) Char const * P; means pointer to a constant.

In case of (a) we con not modify the address of p.
In case of (b) we can not modify the contents of what it points to.

Debugging Exercises

3.1: **What will happen when you execute the following code?**

```
1#include <iostream.h>
2void main()
3{
4    int i=0;
5    i=400*400/400;
6    cout<<i;
7}
```

Ans: i = 400*400/400; Here, 400*400 = 160000 which exceeds the maximum value of int variable. So wrong output will be shown when this program will be run.

Correction :

1Int I = 0;

should be changed as

1long int i = 0;

to see the correct output.

3.2: **Identify the error in the following program.**

```
1include<iostream.h>
2void main()
3{
4    int num[]={1,2,3,4,5,6};
5    num[1]==[1]num ? cout<<"Success" : cout<<"Error";
6}
```

Ans: num [1] = [1] num?. You should write index number after *array name* but here index number is mention before array name in **[1] num**

So expression syntax error will be shown.

Correction : num[1] = num[1]? is the correct format

3.3: **Identify the errors in the following program.**

```
1  #include <iostream.h>
2  void main()
3  {
4      int i=5;
5      while(i)
6    {
7      switch(i)
8      {
9      default:
10     case 4:
```

```
11    case 5:
12    break;
13    case 1:
14    continue;
15    case 2:
16    case 3:
17    break;
18    }
19i-;
20 }
21}
```

Ans:

```
1case 1 :
2continue;
```

The above code will cause the following situation:
Program will be continuing while value of i is 1 and value of i is updating. So infinite loop will be created.
Correction: At last line i- should be changed as i–;

3.4: **Identify the errors in the following program.**

```
1  #include <iostream.h>
2  #define pi 3.14
3  int squareArea(int &);
4  int circleArea(int &);
5  void main()
6  {
7       int a-10;
8       cout << squareArea(a) << " ";
9       cout « circleArea(a) « ";
10      cout « a « endl;
11 {
12      int squareArea(int &a)
13  {
14      return a *== a;
15  }
16   int circleArea(int &r)
17  {
18   return r = pi * r * r;
19  }
```

Ans: Assignment operator should be used in the following line:

1return a *==a;

That means the above line should be changed as follows:

1return a *=a;

3.5: **Missing**

3.6: **Find errors, if any, in the following C++ statements.**
(a) long float x;
(b) char *cp = vp; // vp is a void pointer
(c) int code = three; // three is an enumerator
(d) int sp = new; // allocate memory with new
(e) enum (green, yellow, red);
(f) int const sp = total;
(g) const int array_size;
(h) for (i=1; int i<10; i++) cout << i << "/n"; (i) int & number = 100; (j) float *p = new int 1101;
(k) int public = 1000; (l) char name[33] = "USA";

Ans:

No.	Error	Correction
(a)	too many types	float x; or double x;
(b)	type must be matched	char *cp = (char*) vp;
(c)	No error	
(d)	syntax error	int*p = new int [10];
(e)	tag name missing	enum colour (green, yellow, red)
(f)	address have to assign instead of content	int const * p = &total;
(g)	C++ requires a const to be initialized	const int array-size = 5;
(h)	Undefined symbol i	for (int I = 1; i <10; i++) cout << i << "/n";
(i)	invalid variable name	int number = 100;
(j)	wrong data type	float *p = new float [10];
(k)	keyword can not be used as a variable name	int public1 = 1000;
(l)	array size of char must be larger than the number of characters in the string	char name [4] = "USA";

Programming Exercises

3.1: **Write a function using reference variables as arguments to swap the values of a pair of integers.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3
4  void swap_func(int &a,int &b)
5  {
6
7      cout<<" Before swapping "<<endl
8          <<" a = "<<a<<endl<<" b = "<<b<<endl<<endl;
9      int temp;
10     temp=a;
11     a=b;
12     b=temp;
13     cout<<" After swapping "<<endl
14         <<" a = "<<a<<endl<<" b = "<<b<<endl<<endl;
15
16}
17
18int main()
19{
20     int x,y;
21     cout<<"  Enter two integer value : "<<endl;
22     cin>>x>>y;
23     swap_func (x,y);
24     return 0;
25}
```

output

```
Enter two integer value : 56 61
Before swapping
a = 56
b = 61
After swapping
a = 56
b = 61
```

3.2: **Write a function that creates a vector of user given size M using new operator.**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  int main()
4  {
5     int m;
6     int *v;
7     cout<<"  Enter vector size : "<<endl;
8     cin>>m;
9     v=new int [m];
10    cout<<" to check your performance  insert "<<m<<" integer value"<<endl;
11    for(int i=0;i<m;i++)
12    {
13       cin>>v[i];
14    }
15    cout<<" Given integer value are :"<<endl;
16    for(i=0;i<m;i++)
17    {
18
19       if(i==m-1)
20       cout<<v[i];
21       else
22       cout<<v[i]<<",";
23
24    }
25    cout<<endl;
26    return 0;
27}
```

output

```
Enter vector size : 5
to check your performance insert 5 integer value
7 5 9 6 1
Given integer value are :
7, 5, 9, 6, 1
```

3.3: **Write a program to print the following outputs using for loops**

```
1
22
333
4444
```

55555
……

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  int main()
4  {
5     int n;
6     cout<<" Enter your desired number :"<<endl;
7     cin>>n;
8     cout<<endl<<endl;
9     for(int i=1;i<=n;i++)
10    {
11       for(int j=1;j<=i;j++)
12       {
13          cout<<i;
14       }
15       cout<<endl;
16    }
17    return 0;
18}
```

output

```
Enter your desired number : 6
1
22
333
4444
55555
666666
```

3.4: **Write a program to evaluate the following investment equation**
$V = P(1+r)^n$
and print the tables which would give the value of V for various
of the following values of P, r and n:
P: 1000, 2000, 3000,……,10,000
r: 0.10, 0.11, 0.12,……..,0.20
n: 1, 2, 3,……..,10
(Hint: P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as
$V = P(1 + r)$
$P = V$
In other words, the value of money at the end of the first year becomes the principal amount for the next year and so on)

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  #include<math.h>
4  #define  size 8
5
6  int main()
7  {
8     float v,pf;
9     int n=size;
10    float p[size]={1000,2000,3000,4000,5000,6000,7000,8000};//9000,1000};
11    float r[size]={0.11,0.12,0.13,0.14,0.15,0.16,0.17,0.18};//,0.19,0.20};
12
13    cout<<setw(5)<<"n=1";
14       for(int i =2;i<=size;i++)
15    cout<<setw(9)<<"n="<<i;
16    cout<<"\n";
17
18    for(i=0;i<size;i++)
19    {
20       cout<<setw(-6)<<"p=";
21       for(int j=0;j<size;j++)
22       {
23         if(j==0)
24          pf=p[i];
25
26          v=pf*(1+r[i]);
27
28          cout.precision(2);
29                cout.setf(ios::fixed, ios::floatfield);
30          cout<<v<<setw(10);
31            pf=v;
32       }
33       cout<<"\n";
34
35    }
36    return 0;
37}
```

output

```
n=1         n=2         n=3         n=4         n=5         n=6         n=7
p=1110    1232.1     1367.63     1518.07     1685.06     1870.41     2076.16
p=2240   2508.8    2809.86     3147.04    3524.68     3947.65    4421.36
p=3390   3830.7    4328.69     4891.42     5527.31     6245.86    7057.82
p=4560   5198.4    5926.18      67 55.84    7701.66     8779.89    10009.08
p=5750    6612.5    7604.37     8745.03     10056.79    11565.3     13300.1
p=6960   8073.6    9365.38    10863.84    12602.05   14618.38   16957.32
```

```
p=8190   9582.3    11211.29     13117.21     15347.14    17956.15   21008.7
p=9440   11139.2   13144.26    15510.22     18302.06   21596.43  25483.79
```

3.5: **An election is contested by five candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and count the vote cast for each candidate using an array variable count. In case, a number read is outside the range 1 to 5, the ballot should be considered as a "spoilt ballot" and the program should also count the numbers of "spoilt ballots".**

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  int main()
4  {
5      int count[5];
6      int test;
7      for(int i=0;i<5;i++)
8      {
9         count[i]=0;
10     }
11     int spoilt_ballot=0;
12     cout<<" You can vot candidate 1 to 5 "<<endl
13      <<" press 1 or 2 or 3 or 4 or 5 to vote "<<endl
14      <<" candidate 1 or 2 or 3 or 4 or 5 respectively "<<endl
15      <<" press any integer value outside the range 1 to 5 for NO VOTE    "<<endl<<" press any
16negative value to terminate  and see result :"<<endl;
17
18     while(1)
19        {
20           cin>>test;
21
22           for(int i=1;i<=5;i++)
23           {
24              if(test==i)
25              {
26                 count[i-1]++;
27              }
28           }
29           if(test<0)
30              break;
31         else if(test>5)
32              spoilt_ballot++;
33        }
34        for(int k=1;k<=5;k++)
35        cout<<" candidate "<<k<<setw(12);
36        cout<<endl;
37        cout<<setw(7);
```

```
38      for(k=0;k<5;k++)
39      cout<<count[k]<<setw(13);
40      cout<<endl;
41      cout<<" spoilt_ballot "<<spoilt_ballot<<endl;
42      return 0;
   }
```

output

```
You can vot candidate 1 to 5
press 1 or 2 or 3 or 4 or 5 to vote
candidate 1 or 2 or 3 or 4 or 5 respectively
press any integer value outside the range 1 to S for NO VOTE
press any negative value to terminate and see result :
1
1
1
5
4
3
5
5
2
1
3
6
-1
candidate 1 candidate 2 candidate 3 candidate 4 candidate S
4   1   2   1   3
spoilt_ballot 1
```

3.6: **A cricket has the following table of batting figure for a series of test matches:**

Player's name	Run	Innings	Time not out
Sachin	8430	230	18
Saurav	4200	130	
.	.	.	.

Write a program to read the figures set out in the above forms, to calculate the batting arranges and to print out the complete table including the averages.

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3
4  char *serial[3]={" FIRST "," SECOND " ," THIRD "};//global declaration
```

```
5
6  int main()
7  {
8     int n;
9      char name[100][40];
10    int *run;
11    int *innings;
12    int *time_not_out;
13    cout<<" How many players' record would you insert ? :";
14    cin>>n;
15    //name=new char[n];
16    run=new int[n];
17    innings=new int[n];
18    time_not_out=new int[n];
19
20    for(int i=0;i<n;i++)
21    {
22       if(i>2)
23       {
24          cout<<"\n Input details of "<<i+1<<"th"<<" player's"<<endl;
25       }
26       else
27       {
28          cout<<" Input details of "<<serial[i]<<"player's : "<<endl;
29       }
30
31          cout<<" Enter name : ";
32
33          cin>>name[i];
34          cout<<" Enter run : ";
35          cin>>run[i];
36          cout<<" Enter innings : ";
37          cin>>innings[i];
38          cout<<" Enter times not out : ";
39          cin>>time_not_out[i];
40    }
41
42       float *average;
43       average=new float[n];
44       for(i=0;i<n;i++)
45       {
46          float avrg;
47          average[i]=float(run[i])/innings[i];
48
49       }
50       cout<<endl<<endl;
51       cout<<setw(12)<<"player's name "<<setw(11)<<"run"<<setw(12)<<"innings"<<setw(16)<<"Average"<<setw
52out"<<endl;
53       for(i=0;i<n;i++)
54       {
55          cout<<setw(14)<<name[i]<<setw(11)<<run[i]<<setw(9)<<innings[i]<<setw(18)<<average[i]<<setw(15)<<
```

```
56      }
57      cout<<endl;
58
59      return 0;
   }
```

output

```
How many players record would you insert ? :2
Input details of FIRST player's :
Enter name : Sakib-Al-Hassan
Enter run : 1570
Enter innings : 83
Enter times not out : 10
Input details of SECOND player's :
Enter name : Tamim
Enter run : 2000
Enter innings : 84
Enter times not out : 5
player's name run innings Average times not out
Sakib-Al-Hassan 1570 83 18.915663 10
Tamim 2000 84 23.809525 5
```

3.7: **Write a program to evaluate the following function to 0.0001% accuracy**

(a) $\sin x = x - x^3/3! + x^5/5! - x^7/7! + \ldots\ldots\ldots$

(b) $SUM = 1+(1/2)^2 + (1/3)^3 +(1/4)^4 + \ldots\ldots\ldots$

(c) $\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots\ldots\ldots$

Solution (a):

```
1  #include<iostream.h>
2  #include<math.h>
3  #include<iomanip.h>
4  #define accuracy 0.0001
5  #define pi 3.1416
6
7  long int fac(int a)
8  {
9      if(a<=1)
10         return 1;
11     else
12         return a*fac(a-1);
13}
14int main()
```

```
15{
16   float y,y1,x,fx;
17   int n=1;
18   int m;
19   //const float pi=3.1416;
20   cout<<" Enter the value of angle in terms of degree:  ";
21      cin>>x;
22      float d;
23      d=x;
24   int sign;
25      sign=1;
26if(x<0)
27{
28      x=x*(-1);
29      sign=-1;
30 }
31again:
32   if(x>90 && x<=180)
33   {
34
35      x=180-x;
36
37   }
38   else if(x>180 && x<=270)
39   {
40      x=x-180;
41      sign=-1;
42   }
43   else if(x>270 && x<=360)
44   {
45      x=360-x;
46      sign=-1;
47   }
48
49   else if(x>360)
50   {
51      int m=int(x);
52      float fractional=x-m;
53      x=m%360+fractional;
54      if(x>90)
55         goto again;
56      else
57      sign=1;
58
59   }
60   x=(pi/180)*x;
61   m=n+1;
62   fx=0;
63   for(;;)
64   {
65        long int h=fac(n);
```

```
66          y=pow(x,n);
67          int factor=pow(-1,m);
68          y1=y*factor;
69          fx+=y1/h;
70          n=n+2;
71          m++;
72          if(y/h<=accuracy)
73             break;
74    }
75
76    cout<<"sin("<<d<<")= "<<fx*sign<<endl;
77    return 0;
78}
```

output

```
Enter the value of angle in terms of degree: 120
sin(120)= 0.866027
```

Solution (b):

```
1  #include<iostream.h>
2  #include<math.h>
3  #define accuracy 0.0001
4  int main()
5  {
6     int n;
7     float sum,n1,m;
8     n=1;sum=0;
9     for(int i=1;;i++)
10    {
11          n1=float(1)/n;
12          m=pow(n1,i);
13          sum+=m;
14          if(m<=accuracy)
15            break;
16
17          n++;
18          }
19    cout<<sum<<"\n";
20    return 0;
21}
22       Sample Output(b)
23
24         Solution: (c)
25        #include<iostream.h>
26#include<math.h>
27#define accuracy 0.0001
```

```
28
29      long int fac(int n)
30{
31   if(n<=1)
32      return 1;
33   else
34      return n*fac(n-1);
35}
36
37      int main()
38{
39
40     float y,y1,x,fx;
41     int n=1;
42     int m;
43     const float pi=3.1416;
44     cout<<" Enter the value of angle in terms of degree:  ";
45     cin>>x;
46     if(x<0)
47       x=x*(-1);
48     x=(pi/180)*x;
49
50       fx=1;
51
52        m=2;
53       float y2;
54       long int h;
55       for(;;)
56       {
57          h=fac(m);
58          int factor=pow(-1,n);
59          y1=pow(x,m);
60          y2=(y1/h)*factor;
61          fx+=y2;
62          if(y1/h<=accuracy)
63             break;
64          m=m+2;
65          n++;
66       }
67       cout<<fx<<"\n";
68}
```

output

```
Enter the value of angle in terms of degree: 60
0.866025
```

3.8: **Write a program to print a table of values of the function**

$Y = e^{-x}$

For x varying from 0 to 10 in steps of 0.1. The table should appear as follows

TABLE FOR Y =EXP[-X];

X 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.900

1.0

.

.

9.0

Solution:

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  #include<math.h>
4          int main()
5  {
6         float x,y;
7         cout<<"                    TABLE FOR  Y=EXP(-X)          :\n\n";
8         cout<<"x";
9         for(float k=0;k<.7;k=k+0.1)
10        cout<<setw(10)<<k;
11        cout<<"\n";
12        for(k=0;k<10*.7;k=k+0.1)
13        cout<<"-";
14        cout<<"\n";
15            for(float j=0;j<10;j++)
16        {
17           cout<<j<<setw(4);
18           for(float i=0;i<.7;i=i+0.1)
19           {
20                 x=i+j;
21                 y=exp(-x);
22                 cout.precision(6);
23                              cout.setf(ios::fixed,ios::floatfield);
24                 cout<<setw(10)<<y;
25           }
```

```
26          cout<<"\n";
27      }
28      return 0;
29}
```

Note: Here we work with 0.4 for a good looking output.

output

TABLE FOR Y=EXP(-X)

x	0	0.1	0.2	0.3	0.4
0	1	0.904837	0.818731	0.740818	0.67032
1	0.367879	0.332871	0.301194	0.272532	0.246597
2	0.135335	0.122456	0.110803	0.100259	0.090718
3	0.049787	0.045049	0.040762	0.036883	0.033373
4	0.018316	0.016573	0.014996	0.013569	0.012277
5	0.006738	0.006097	0.005517	0.004992	0.004517
6	0.002479	0.002243	0.002029	0.001836	0.001662
7	0.000912	0.000825	0.000747	0.000676	0.000611
8	0.000335	0.000304	0.000275	0.000249	0.000225
9	0.000123	0.000112	0.000101	0.000091	0.000083

3.9: **Write a program to calculate the variance and standard deviation of N numbers**

Variance $=1/N \sum(x_i - x)^2$

Standard deviation$=\sqrt{1/N \sum(x_i - x)^2}$

Where $x = 1/N \sum x_i$

Solution:

```
1  #include<iostream.h>
```

```
2  #include<math.h>
3          int main()
4  {
5      float *x;
6              cout<<" How many number ? :";
7      int n;
8      cin>>n;
9      x=new float[n];
10     float sum;
11     sum=0;
12     for(int i=0;i<n;i++)
13     {
14           cin>>x[i];
15           sum+=x[i];
16     }
17     float mean;
18     mean=sum/n;
19     float v,v1;
20     v1=0;
21     for(i=0;i<n;i++)
22     {
23           v=x[i]-mean;
24           v1+=pow(v,2);
25     }
26     float variance,std_deviation;
27     variance=v1/n;
28     std_deviation=sqrt(variance);
29     cout<<"\n\n variance = "<<variance<<"\n standard deviation = "<<std_deviation<<"\n";
30
31    return 0;
32}
```

output

```
How many number ? :5
10
2
4
15
2
variance = 26.24
standard deviation = 5.122499
```

3.10: **An electricity board charges the following rates to domestic users to**

discourage large consumption of energy:

For the first 100 units – 60P per unit

For the first 200 units – 80P per unit

For the first 300 units – 90P per unit

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added.

Write a program to read the names of users and number of units consumed and print out the charges with names.

Solution:

```
#include<iostream.h>
#include<iomanip.h>
int main()
{
   int unit;
   float charge,additional;
   char name[40];
   while(1)

   {
      input:
          cout<<" Enter consumer name & unit consumed :";
           cin>>name>>unit;
           if(unit<=100)
           {
              charge=50+(60*unit)/100;
           }
           else if(unit<=300 && unit>100)
           {
              charge=50+(80*unit)/100;
           }
           else if(unit>300)
           {
              charge=50+(90*unit)/float(100);
              additional=(charge*15)/100;
              charge=charge+additional;
           }
           cout<<setw(15)<<"Name"<<setw(20)<<"Charge"<<endl;
           cout<<setw(15)<<name<<setw(20)<<charge<<endl;
           cout<<" Press o for exit / press 1 to input again :";
           int test;
           cin>>test;
           if(test==1)
           goto input;
           else if(test==0)
           break;
   }
```

```
38    return 0;
39}
```

output

```
Enter consumer name & unit consumed :sattar 200
Name                Charge

sattar              210

Press o for exit / press 1 to input again :1

Enter consumer name & unit consumed :santo 300

Nmae          Charge

santo         290

Press o for exit / press 1 to input again : 0
```