OBJECT ORIENTED PROGRAMMING IN C++

Module 1

Programming Techniques

We can distinguish the programming techniques into following categories.

- Unstructured programming
- Structured programming
- Modular programming
- Top – Down Programming
- Bottom – Up Programming
- Object-oriented programming

Unstructured Programming : Usually, people start learning programming by writing small and simple programs consisting only of one main program. Here “main program" stands for a sequence of commands or *statements* which modify data which is *global* throughout the whole program. We can illustrate this as shown in Fig. 1. This programming technique provides tremendous disadvantages once the program gets sufficiently large. For example, if the same statement sequence is needed at different locations within the program, the sequence must be copied.

Figure 1:

Unstructured programming. The main program directly operates on global data.

Structured Programming : Structured programming is a programming method you use to break long programs into numerous small procedures. With procedural programming you are able to combine returning sequences of statements into one single place. A procedure call is used to invoke the procedure. After the sequence is processed, flow of control proceeds right after the position where the call was made. See Fig. 2.

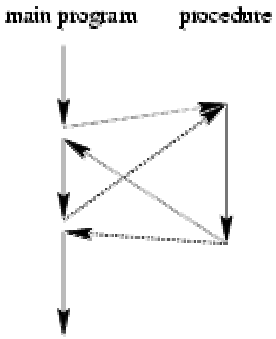

Figure 2

Execution of procedures. After processing flow of controls proceed where the call was made.

With introducing parameters as well as procedures of procedures (sub procedures) programs can now be written more structured and error free. For example, if a procedure is correct, every time it is used it produces correct results. Consequently, in cases of errors you can narrow your search to those places which are not proven to be correct.

Now a program can be viewed as a sequence of procedure calls. The main program is responsible to pass data to the individual calls, the data is processed by the procedures and, once the program has finished, the resulting data is presented. Thus, the flow of data can be illustrated as a hierarchical graph, a tree, as shown in Fig. 3 for a program with no sub procedures.

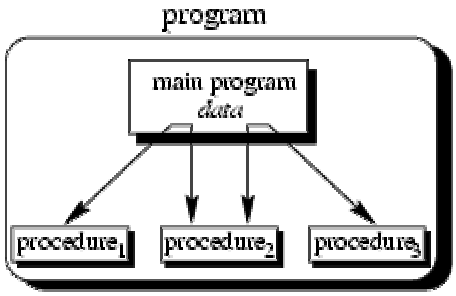

Figure 3:
Procedural programming. The main program coordinates calls to procedures and hands over appropriate data as parameters.

Advantages of Structured Programming : The goal of structured programming is to create correct programs that are easy to write, understand and change.

1. Easy to write:
    - Structured design increases the programmer's productivity by allowing them to look at the big picture first and focus on details later.
    - Several Programmers can work on a single, large program, each working on a different module
    - Studies show structured programs take less time to write than standard programs.
    - Procedures written for one program can be reused in other programs requiring the same task. A procedure that can be used in many programs is said to be reuseable
2. Easy to debug

    Since each procedure is specialized to perform just one task, a procedure can be checked individually. Older unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. The logic of such programs is cluttered with details and therefore difficult to follow.
3. Easy to Understand
    - The relationship between the procedures shows the structured design of the program.
    - Meaningful procedure names and clear documentation identify the task performed by each module/
    - Meaningful variable names help the programmer identify the purpose of each variable.
4. Easy to Change

    Since a correctly written structured program is self-documenting, it can be easily understood by another programmer.

Modular Programming : With modular programming procedures of a common functionality are grouped together into separate *modules*. A program therefore no longer consists of only one single part. It is now divided into several smaller parts which interact through procedure calls and which form the whole program (Fig. 4).

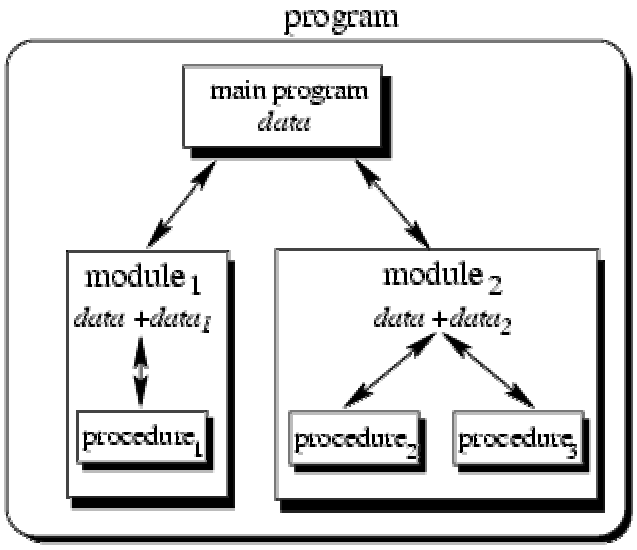

Figure 4: Modular programming. The main program coordinates calls to procedures in separate modules and hands over appropriate data as parameters.

Each module can have its own data. This allows each module to manage an internal *state* which is modified by calls to procedures of this module. However, there is only one state per module and each module exists at most once in the whole program.

Characteristics of Modular Programming

1) Modular programming is one way of managing the complexity of larger programs
2) The division of programs into modules is a powerful organizing principle for designing non-trivial programs
3) Modular programming groups related sets of functions together into a module.
4) The module is divided into an interface and an implementation.
5) The two most important elements of a module are the division of the module into an interface and an implementation and the ability to hide information in the implementation
6) The implementation of the module is private and hidden from the users
7) Modules provide abstraction, encapsulation, and information-hiding, making the large-scale structure of a program easier to understand.
8) Careful design of modules also promotes software reuse, even in embedded systems programming.

Top-down programming : Top-down programming refers to a style of programming where an application is constructed starting with a high-level description of what it is supposed to do, and breaking the specification down into simpler and simpler pieces, until a level has been reached that corresponds to the primitives of the programming language to be used.

Top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures. Typically, the high-level specification of the system states functionality. This high-level description is then refined to be a sequence or a loop of simpler functions or procedures, that are then themselves refined, etc. In this style of programming, there is a great risk that implementation details of many data structures have to be shared between modules, and thus globally exposed. This in turn makes it tempting for other modules to use these implementation details, thereby creating unwanted dependencies between different parts of the application.

Disadvantages of top-down programming

Top-down programming complicates testing. Noting executable exists until the very late in the development, so in order to test what has been done so far, one must write stubs(Pieces of programs, usually functions or procedures) .

Furthermore, top-down programming tends to generate modules that are very specific to the application that is being written, thus not very reusable.

But the main disadvantage of top-down programming is that all decisions made from the start of the project depend directly or indirectly on the high-level specification of the application. It is a well-known fact that this specification tends to change over time. When that happens, there is a great risk that large parts of the application need to be rewritten.

Bottom-up programming : Bottom-up programming refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until all of the application has been written.

In a language such as C or Java, bottom-up programming takes the form of constructing abstract data types from primitives of the language or from existing abstract data types.

Advantages of bottom-up programming

Bottom-up programming has several advantages over top-down programming.

Testing is simplified since no stubs are needed. While it might be necessary to write test functions, these are simpler to write than stubs, and sometimes not necessary at all, in particular if one uses an interactive programming environment such as Common Lisp or GDB.

Pieces of programs written bottom-up tend to be more general, and thus more reusable, than pieces of programs written top-down. In fact, one can argue that the purpose bottom-up programming is to create an application-specific language. Such a language is suitable for implementing an entire class of applications, not only the one that is to be written. This fact greatly simplifies maintenance, in particular adding new features to the application. It also makes it possible to delay the final decision concerning the exact functionality of the application. Being able to delay this decision makes it less likely that the client has changed his or her mind between the establishment of the specifications of the application and its implementation.

Object – Oriented Programming :

A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.

In structured languages, a user defined data type specifies only the structure of the variables of that type. It does not specify the operations that are to be performed on that type. For example, in C a declaration of a struct specifies the composition of data items, but it is left to the user program, to decide the type of operations that are to be performed upon variables of that type.

A collection of data and its operations is referred to as and ‘object’. The data is local to an object. This is encapsulated within an object and is not accessible from outside the object. Thus, other external objects cannot access the data residing within an object. These objects only know how to interact with an other object through the interface of its operation.

Advantages of OOP : Many problems encountered earlier, are resolved with the definition of an object. The notion of an object has certain properties which give rise to many advantages. These are:

a) No accidental modification of data
b) High specification level
c) Easier maintenance
d) Easier Upgradation
e) High Level of Modularity
f) Reuse

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify.

Need for Object Oriented Programming :

Until recently, programs were envisioned as a series of procedures that acted on data. A procedure, or function, was defined as a set of specific instructions executed in sequential manner. The data is kept separate from the procedures, and the trick in programming was to keep track of, which functions called which functions, and which data was changed. To make sense of this potentially confusing situation, structured programming was created. Pascal is a language, which uses procedural method of programming.

The principal idea behind structured programming was as simple as the idea of “divide and conquer.” A computer program could be regarded as consisting of a set of tasks. Any task that was too complex to be simply described would be broken into a set of smaller component tasks, until the tasks were sufficiently small and self-contained to be easily understood. The language C uses structured programming approach.

Structured programming remains an enormously successful approach for dealing with complex problems. By the late 1980’s, however, some of its deficiencies had become all too clear.

First, the separation of data from the tasks that manipulate the data became increasingly difficult to comprehend and maintain. It is natural to think of data (employee records, for example) and ways that data can be manipulated (sort, edit, and so on) as related ideas.

Second, programmers found themselves constantly reinventing new solutions to old programs. Reinventing in a way is opposite of reusability. Reusability involves building components that have known properties, and then plugging those components into a program, as the programmer needs them. This software concept is modeled after the hardware world; when an engineer needs a new transistor, she doesn’t usually invent one but goes to big bin of transistors and finds one that fits her needs, or perhaps modifies the found transistor. No similar option existed for a software engineer until reusability was developed.

Object-oriented programming (OOP) attempts to meet these needs, providing techniques for managing enormous complexity, achieving reuse of software components, and coupling data with the tasks that manipulate the data.

The essence of object-oriented programming is to treat data and the procedures that act on the data as a single “object” – a self-contained entity with an identity and certain characteristics of its own.

Applications OOP

Object-oriented programming is not particularly concerned with the details of the program operation. Instead, it deals with the overall design of the program . The OOP features can be implemented in many languages that support them. C++ is one language that supports all the features of OOP. Other languages include Common Lisp object system, Smalltalk, Eiffel, Actor and the latest version of Turbo Pascal, Java. However, C++ is the most widely used object-oriented programming language.

Applications of OOP are beginning to gain importance in many areas. The most popular application of object – oriented programming, has been in the area of user interface design such as windows. Real – business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of application because it can simplify a complex problem. The promising areas for aplication of OOP include :

1. Real – Time Systems
2. Simulation and Modelling
3. Object – Oriented Databases
4. AI and Expert systems
5. Neural Networks and parallel programming
6. Decision support and office automation systems
7. CAD Systems

Characteristics of Obect Oriented Programming

C++ fully supports object-oriented programming, including the four pillars of object-oriented development :

- Encapsulation
- Data hiding
- Inheritance
- Polymorphism

Other important features are

- Data Abstraction
- Dynamic Binding
- Communication

1. **Encapsulation and Data Hiding** : The property of being a self-contained unit is called encapsulation. The idea that the encapsulated unit can be used without knowing how it works is called data hiding.

   Encapsulation is the principle by which related contents of a system are kept together. It minimizes traffic between different parts of the work and it separates certain specific requirements from other parts of specification, which use those requirements.

   The important advantage of using encapsulation is that it helps minimize rework when developing a new system. The part of the work, which is prone to change, can be encapsulated together. Thus any changes can be made without affecting the overall system and hence changes can be easily incorporated.

   As noted before, when an engineer needs to add resistor to the device she is creating, she doesn't typically build a new one from the scratch. She walks over to a bin of resistors, examines the bin of resistors, examines the colored bands that indicate the properties, and picks the one she needs. The resistor is a "black box" as far as the engineer is concerned – that is, she doesn't care how it does its work as long as it conforms to her specifications. All the resistor's properties are encapsulated in the resistor object – they are not spread out through the circuitry. It is not necessary to understand how the resistor works to use it effectively, because its data is hidden inside the resistors casing.

   C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. Once created, a well-defined class acts as a fully encapsulated entity and can be used

as a whole unit. The actual inner workings of the class should be hidden; users of well-defined class do not need to know how the class works, only how to use it.

2. **Inheritance and Reuse** : Inheritance is the process by which objects of one class acquire the properties of objects of another class. In OOP, the concept of inheritance provides the idea of resusability. This means we can add additional features to an existing class without modifying it.The new class will have the combined features of both the classes.

   Consider a car manufacturing company. When they want to build a new car, they have two choices. They can start from the scratch, or they can modify an existing model. Perhaps their former model is nearly perfect (say CAR1), but they would like to add a turbocharger and a six-speed transmission. The chief engineers would prefer to use the earlier model and make some changes to it than creating a new one (CAR2).

   C++ supports the idea of reuse through inheritance. Through this concept, a new type can be declared that is an extension of the existing type. This new subclass is said to derive from the existing type and is sometimes called a derived type.

3. **Polymorphism** : Polymorphism means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

   The CAR2 in the above example may respond differently than the CAR1 when the car accelerates. The CAR2 might engage fuel injection and a turbocharger, for example, whereas the CAR1 simply get petrol into its carburetor. A user, however, does not have to know about these differences; the user simply presses the accelerator and the car responds with the correct function.

   C++ supports this idea – that different objects do “the right thing “ – through function overloading and operator overloading. Using a single function name to perform different types of tasks is known as function overloading. The process of making an operator to exhibit different behaviours in different instances is known as operator overloading. Poly means many, whereas morph means form. Thus, polymorphism refers to the same name taking many forms.

4. **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 list of abstrat 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 members because they hold information. The functions that operate on these data are sometimes called methods or member functions. Since classes use the concept of data abstraction, they are known as Abstract Data Types.

5. **Dynamic Binding** : Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run – time. It is associated with polymorphism and inheritance.

6. **Communication** : Communicating with messages is the principle for managing complexity, which is used when interfaces are involved to communicate between the entities. OOP uses classes and objects have attributes(data members), which can be exclusively accessed by services(member functions) of the objects. To change the attributes of the object (a data member of the object), a message has to be sent to the object to invoke the service. This mechanism helps in implementing encapsulation and data abstraction.

Basic Concepts and Terminology

There are several differences between the C language and C++, which have nothing to do with OOP. Some of them are highlighted below.

Additional keywords in C++

class	friend	virtual	inline
private	public	protected	const
this	new	delete	operator

The actual use and description of these additional keywords will be covered in their specific contexts.

Additional Features in C++

1. Comments : Comments are integral part of any program . Comments help in coding, debugging and maintaining a program . The compiler ignores them. They should be used liberally in the program .

In C++, a comment starts with two forward slashes (//) and ends with the end of that line. A comment can start at the beginning of the line or on a line following the program statement. This form of giving comments is particularly useful for the short line comments. If a comment continues on more than a line, the two forward slashes should be given on every line.

The C style of giving a comment is also available in C++. This style (/*….*/) is particularly useful when the comment spans more than a line.

e.g.

```
void main()
{
                    /* this is a good old style of giving a comment. It can be
                    continued to next line and
                    has to be ended with */
        clrscr();       // Clears the Screen
    init()_;   // Initialize variables.
    :
    :
 }
```

2. Variable Declaration : This declaration of variables in the C language is allowed only in the beginning of their block, prior to executable program statements. In C++ declaration of variables can be interspersed with executable program statements. The scope id variables, however, remains the same – the block in which they are declared.

e.g.

```
void main()
{
    int x = 10;
    printf (" the value of x= % d\n",x);
    int y = 0;
    for( int z= 0; z < 10; ; z++)          // variable declared here.
    {
            y ++;
            x ++;
    }
```

```
}
```

Although, a deviation from the old style of declaring all variables in the beginning of the block, this does save some amount of memory, i.e., a variable is not given a memory until the declaration statement. Also, since a variable can be declared just before using it is suppose to give a better control over variables.

3. The Scope Resolution Operator(::) : Global variables are defined outside any functions and thus can be used by all the functions defined thereafter. However, if a global variable is declared with the same name as that of a local variable of a function , the local variable is the one in the scope when the program executes that function . The C++ language provides the scope resolution operator (::) to access the global variable thus overriding a local variable with the same name. This operator is prefixed to the name of the global variable . The following example shows its usage.

```
int global = 10;

void main()
 {
       int global = 20;
       printf(" Just writing global prints : %d\n", global);
           printf(" Writing ::global prints : %d\n", ::global);
 }
```

The output of this program will be:

```
Just writing global prints : 20
Writing ::global prints    : 10
```

4. Default Arguments : A default argument is a value that is automatically assigned to a formal variable, if the actual argument from the function call is omitted.

e.g.

```
void drawbox( int x1=1, int y1=1, int x2=25, int y2=80, int color=7);      // Prototype

void main ( void )
{
       drawbox(10,1,25,80,14);                                              //
parameters passed
       drawbox();                                                           //        uses
default arguments      }

void drawbox( int x1, int y1, int x2, int y2, int color)
{
          // Body of the function
}
```

Function drawbox() draws a box around the edges of the coordinates passed as parameters. If these parameters are omitted, then the default values, as given in the declaration are passed.

When a function is declared, default values must be added from right to left. In other words, a default value for a particular argument cannot be given unless all default values for the arguments to its right are given. This is quite

logical, since, if there are some arguments missing in the middle then the compiler would not know as to which arguments have been specified and which should be taken as default.

e.g.

```
void drawbox( int x1=1, int y1=1, int x2=25, int y2=80, int color = 7);     // Valid

void drawbox( int x1, int y1, int x2, int y2, int color = 7);                //
Valid

void drawbox( int x1=1, int y1=1, int x2=25, int y2=80, int color );          // Not Valid
```

Default arguments are useful when the arguments almost have the same value. They are also useful to increase the number of arguments to a function in an already working program. In this case, using default arguments mean that existing function calls need not be modified, whereas, new function calls pass more number of arguments.

5. The new and delete operators : The C language has defined library functions- malloc() and free() for dynamic allocation and de-allocation of memory. C++ provides yet another approach to allocate blocks of memory – the new operator. This operator allocates memory for a given size and returns a pointer to its starting point. C++ also provides delete, to release the memory allocated by new. The pointer returned by the new operator need not be typecasted.

e.g.

```
char arr[100];                  // Compile_time allocation of an array
char *arr;              // Character pointer

arr = new char[size];  //Run time allocation of an array. Size can be a constant or a
variable.
```

In the above example, new returns a pointer to a block of size bytes. It is synonymous with declaring a character array. However, declaring an array is an example of static binding – the array is built at the compile-time. This array remains in existence right from the beginning of the program to its end, even if not in use. Whereas , the array declared by new operator can be allocated memory only when required and can be released when over with, using the delete operator. This is an example of dynamic binding.

It is possible that the system may not have enough memory available to satisfy a request by the new operator. In this case, new returns a null pointer. Thus, the returned value from the new should always be checked before performing any operations on it, which otherwise may lead to an adverent system crashes.

Once the new operator allocates memory, it can be accessed using the indirection operator (*) or the array subscripts.

e.g.

-
```
int *some = new int [10];          // A block of 10 integers
double *d5 = new   double[5]; // A block of 5 doubles
char *cp;
cp = new char;          // One character only.
delete some;          // Release memory.
delete d5;
```

- char *ptr , *str = “the shy is blue “;
  ptr = new char [strlen(str)+1]; // Character array
  if(ptr != 0)
  {
  strcpy(ptr,str);
  :
  :
  }
  else
  {
  printf(“ not enough memory “);
  }

- struct date
  {
  int dd,mm,yy;
  };

  date *dp = new date; // This allocated a new structure to the pointer variable dp
  printf(“ enter day “);
  scanf(“%d”,&dp->dd);

6. The cin and cout objects : In C++ the keyword cin with the extraction operator “ >> “ is used to accept the input and the keyword cout with the insertion operator “ << “ is used to display string literals and values, in place of scanf() and printf() functions of C. However, here we do not need any type specifiers like %d , %f etc., to denote the variables type. cin and cout can be used in context with any basic data types.

e.g.

```
int ivar;
float fvar;
char cvar;
char name[10];

cin >> ivar ;
cin >> fvar >> cvar >> name;

cout<< “ ivar = “ << ivar << endl;
cout<< “ fvar = “ <<fvar << “ cvar= “ << cvar << name;
```

It is clear in the above example, how we can accept and display variables. We can also concatenate various variables, using the insertion operator and extraction operator. While displaying, if you write some text in the quotes they are displayed as it is. Another manipulator endl can be used to end the line and get the cursor back to next line.

7. Reference Variables : The symbol “&” is interpreted as an address operator as well as AND operator. This operator is also used to declare a “reference variable”. A reference is referred to an alias, serves as an alternative name for the object.

e.g.

```
int ival  = 1024;
int &rval  =  ival;  // rval is reference to ival
```

- A reference variable should be initialized at time of declaration itself.

e.g.

```
int &iref;          // wrong
int &iref  =  ival  //o.k.
```

- Once a reference has been made to a variable , it cannot be changed to refer to another variable.

e.g.

```
void main()
{
        int num1  =  10, num2  =  200;
        int &ref  =  num1;
        cout<< num1 << num2 << ref ;   //10,200,10;
        ref  =  num2;
        cout << num1 << num2 << ref ;  //200,200,200
}
```

It changes the contents of the original variable, which is not the desired action.

- Although a reference to a variable serves as a kind of pointer, we cannot initialize the address of an object to it.

e.g.

```
int ival  =  1000;
int &refval  =  &ival;      //the error message by the compiler would be- can not convert int* to int.
```

However , we can define a pointer to a reference.

e.g.

```
int *pi  =  &ival;
int *&ptr_ref  =  pi;
```

- All the operations on the reference variables are actually applied to the object to which it refers.

e.g.

```
refval  =  refval + 2;      // ival  =  ival + 2 ,1000 + 2 =1002
int ii = refval;            // ii = 1002;
int *pi = &refval           //pi = &ival;
```

- A constant reference can be initialized to an object of a different type as well as to non-addressable values, such as literal constants .

e.g.

```
double dval = 3.1415;
const int &ir = 1024;
const int &ir2 = dval;
const double &dr = dval +1.0;
```

The same initializations are not legal for non-constant references.

Internally, a reference maintains the address of the object for which it is an alias. In case of non-addressable values, such as literal constant and objects of different types, the compiler generates a temporary object that reference actually addresses but user has no access to these addresses.

e.g.

```
double  dval = 1024;
const int &ri = dval;
```

The compiler transforms this as,

```
int temp = dval;
const int &ri = temp;
```

Now if we try to change ri, it actually changes temp and not dval as it is a constant

We can initialise a reference to the address of a constant object.

If we write,

```
const int ival = 1000;
int *&pi_ref = &ival;        // we require constant reference.
```

If we write,

```
const int *&pi_ref = &ival;
```

/* this is reference to a pointer to an object of type int defined to be a constant. Our reference is not to a constant but rather to a non-constant, which addresses a constant object.*/

The correct form is,

```
int *const &pi_ref = &ival  //o.k.
```

Differences between a pointer and a reference

1. A reference must always point to some object where as this restriction is not imposed on a pointer.

   e.g.

   ```
   int *pi = 0;
   ```

   pointer can point to no object. would be converted as ,

   ```
   const int &ri = 0

   int temp = 0;
   const int &ri = temp;
   ```

2. The assignment of one reference with another changes the object being referenced and not the reference itself.

   e.g.

   ```
   int ival1 = 1000, ival2 = 2000;
   int *pi1 = &ival1, *pi2 = &ival2;
   int &ri1 = ival1, &ri2 = ival2;
   pi1 = pi2;
   ```

   - ival1 remains unchanged but pi1 and pi2 now address the same object ival2.

     ```
     ri1 = ri2;
     ```

- ival1 becomes 2000. ri1 and ri2 still refer to ival1 and ival2 respectively.

C++ PROGRAMMING BASICS

Literals (Constants) : Constants are *data storage locations whose address is not accessible* for the user. Their value is not changed during the course of the program.

Literal constant : represented by their value.
Symbolic constants : represented by their symbols / names.

e.g.

```
sum = num1 + 10;

//10 is a literal constant.
// sum is a symbolic constant.
```

Constants as well as variables have types associated with them.

The following table summarizes the data types , their sizes and their range, for a 16-bit machine.

Type	Size in Bytes	Range
Unsigned short int.	2	0 to 65,535
Short int.	2	-32,768 to 32767
Unsigned long int.	4	0 to 4,294,967,295
Long int.	4	-2,147,483,648 to 2,147,483,647
int.	2	-32,768 to 32,767
Unsigned int.	2	0 to 65,535
Char	1	256 character values
Float	4	1.2e-308 to 3.4e38
Double	8	2.2e-308 to 1.8e308

Integer literal : Integer are numbers without fractional parts.

e.g.

```
20      // Decimal
024     // Octal
0x14    // Hexadecimal
```

To denote long, unsigned, suffixes such as L, U, l, u can be used.

e.g.

128U, 1028u, 1L, 8LU, 71l(71 and small L) .

Floating literal : They can be written in *common decimal* as well as *scientific notation(floating point representation). By default it is of type double.* F, L are applied only to common decimal notation.

e.g.

3.1415F , 1.0E-3, 12.345L, 2. , 3E1, 1.0L

Boolean constants : It can hold 2 values – true(1) or false(0).

Character literal : Character literals are characters , digits and some special symbols, which are enclosed in single quotes and require a single byte of memory.

e.g.

'a', '2', '.', ' '(blank) -uses a byte.

Special Character set (Escape Sequences) : When normal characters are preceded with a \ (back slash) , they work in a different way. These are called escape sequences.

e.g.

\n – new line
\t – horizontal tab
\v – vertical tab
\" – double quotes.

They can also be represented using literal constants .

e.g.

\14 instead of \n for newline.

String literal : An array of constant characters is represented as strings. Zero or more characters are represented within double quotation marks. These are terminated by a null character. The compiler does this.

e.g.

" " -- null string
"a"
"\ncc\toptions\tfile.[cc]\n"

" a multiline \
string literal signals its \
continuation with a backslash"

Qualifiers : All variables by default are signed (the sign bit is also stored as a part of the number). If it is known beforehand that the value of a variable will not be signed, then the keyword *unsigned* can precede a variable, thus declaring it an unsigned variable. This is called as a *qualifier.*

e.g.

```
unsigned int num = 341;
unsigned arr = 20;
```

Note that in the second example the data type is omitted, in which case it defaults to being an integer.

There are many occasions when values to be stored in the variable exceeds the storage space allocated to them. For instance, an integer on a 16-bit processor is allocated 2 bytes, which can store at the most 65535, if the variable is unsigned. If a bigger number is to be stored in a variable , then the qualifier *long* can be used.

e.g.

```
long int num = 1234354l;
long emp = 1234321L;
unsigned long  int ulinum = 0;
unsigned long ulnum;
```

Note that the two qualifiers with one variable declaration can also be given. In this case the last two variables can have double the amount of storage and are unsigned. Also note that the letter 'L' or 'l' succeeds long constants.

Similarly, there is another qualifier *short*, which gives storage equal to, or less than an integer, depending upon the implementation.

e.g.

```
short int i;
```

Some implementations have the qualifier *signed*. All the variables by default are signed, therefore this qualifier is not of much use.

Operators : The variables, which are declared and defined, are the operands, which are operated upon by the operators. Operators specify what operations are to be performed on the operands. The language offers a range of operators, ranging from arithmetic, relational and logical operators to bit-wise logical , compound assignment and shift operators.

- Assignment Operators : The equal (=) sign is used for assigning a value to another. The left hand side has to be a variable (lvalue, which excludes constants, expressions, functions etc.).

  e.g.

  ```
  x = 10;
  y = x;
  ```

- Arithmetic Operators : We can classify the arithmetic operators as UNARY and BINARY operators.

  - Unary Operators : Unary operators are those, which operate on a single operand.

    - Unary Minus Operator(Negation) : This operand can be used to negate the value of a variable. It is also used to specify a negative number, here a minus(-) sign is prefixed to the number.

      e.g.

      ```
      int x = 5;
      int z = -10;
      y = -x;
      a = z;
      ```

      The value of y now becomes –5 and 'a' becomes -10. However, the language does not offer any unary + operator.

    - Increment and Decrement Operators : The operator for increment is '++' and decrement is '—'. These operators increase or decrease the value of a variable on which they are operated, by one(1). They can be used as prefix or postfix to the variable, and their meaning changes depending on their usage. When used as prefix, the value of the variable is incremented/ decremented before using the expression. But when used as postfix, its value is first used and then the value is incremented or decremented.

      e.g.

      ```
      --x;
      x--;
      ```

      In the above example it does not matter whether the decrement operator is prefixed or suffixed. It will produce the same result. However, in the following example it does make a difference :

      ```
      int a =0, b=10;
      a = ++b; is different from a=b++;
      ```

      In the first case, the value of 'a' after the execution of this statement will be 11, since 'b' is incremented and then assigned. In the second case, the value of 'a' will be 10 , since it is assigned first and then incremented. The value of 'b' in both the cases will be 11.

The unary operators have a higher precedence than the binary arithmetic operators.

- Binary Operators : There are five binary operators. These operators, require two operands.

'+'	for addition.
'-'	for subtraction.
'*'	fro multiplication.
'/'	for division .
'%'	for modulus.

e.g.

```
z = x + y;
f = 10 / 3;
f =  10 % s;
x = a + b – c / d * e;
x = ((a + (b – c) / d )* e;
```

The second example will give the quotient of 10 divided by 3, where as the third example will give the remainder after division.

The expressions in the fourth example is evaluated according to the precedence of operators, which is as follows:

%, /, *	on the same level, evaluated from left to right.
+, -	on the same level, evaluated from left to right.

Arithmetic expressions can also contain parenthesis to override the precedence, as shown in the last example.

- Compound Assignment Operators : Apart from the binary and the unary arithmetic operators, we also have compound assignment operators. These are +=, -=, *=, /=, %=. Using these operators, the expression

```
x = x + 5;
```

can also be written as

```
x += 5;
```

This helps in writing compact code.

- Relational Operators : A relational operator is used to make comparison between two values. All these operators are binary and require two operands. There are the following relational operators :

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

e.g.

```
number < 6
ch != 'a'
total == 1000
```

Note that no space is given between the symbols. All these operators have equal precedence amongst themselves and a lower precedence than the arithmetic operators.

- Logical Operators : We say any expression that evaluates to zero is a FALSE logic condition and that evaluating to non-zero value is a TRUE condition. Logical operators are useful in combining one or more conditions. The following are the logical operators :

  && AND
  || OR
  ! NOT

  The first two operators are binary, whereas the exclamation(!) is a unary operator. It is used to negate the condition.

  e.g.

  x<0 && y>10 evaluates to true if both conditions are true
  x < 0 || z = = 0 evaluates to true, if one of the conditions is true
  !(x == 0) evaluates to true if condition is false.

  The unary negation operator(!) has a higher precedence amongst the these, followed by the and (&&) operator and then the or(||) operator, and are evaluated from left to right.

- Bit-wise Operators : Some applications require operations to be done on different bits of a byte separately. Bit-wise operators offer a facility to do just that. There are several bit-wise operators :

  - Unary Operator : One's Complement Operator(~) : This operator is a unary operator that causes the bits of its operand to be inverted, i.e. 1 becomes 0 and 0 becomes 1. for instance, to see the largest possible number, which can be stored in an unsigned integer, a zero can be assigned to it. All the bits in this word will be zeros. When the one's complement operator is used on this word, all the bits will be inverted to ones, giving the largest possible number. The program to show this conversion is below :

```
main()
{
        unsigned u = 0;
        printf("Value before conversion : %d\n",u);
        u = ~u;
        printf("Value After conversion : %d\n",u);
}
```

  - Binary logical bit-wise operators : There are three logical bit-wise operators :

    & and
    | or
    ^ exclusive or

    These are binary operators. The operations are carried out independently on each pair of the corresponding bits of the operands. i.e. the bit 1 of operand 1 is logically operated with the bit 1 of operand 2. the operations of these operators are summarized in the following table:

  e.g.

```
char x, y, z, a, b;

a = 0x01;       /* 0000 0001 */
b = 0x05;       /* 0000 0101 */
```

```
x = a & b;
y = a ^ b;
z = a | b;
x &= y;
y |= z;
```

The result of x is 0x01, since

```
a =    0 0 0 0 0 0 0 1
       & & & & & & & &
b =    0 0 0 0 0 1 0 1
-------------------------
x =    0 0 0 0 0 0 0 1
-------------------------
```

- The Shift Operators : There are two shift operators : left shift (<<) and right shift (>>). These are binary operators. The format is

  operand >> number **or** operand << number

  The first operand is the value, which is to be shifted. The second is the number of bits by which it is shifted. The left shift operators shift 'number' bits to the left, whereas the right shift operators shift 'number' bits to the right. The leftmost and the rightmost bits are shifted out and are lost.

e.g.

```
char x, y = 0x80;

y = y >> 3;
x = y <<2;
y <<= 1;
x >>=2;
```

The char y(which is allocated a byte of space on a 16-bit machine) will have its bits shifted to the right three places. The value of y which was 0x80(1000 0000) is changed to 0xFo(1111 0000) , after y = y>>3.

Precedence and Order of evaluation : The languages follow a standard precedence for basic operators. Precedence rules help in removing ambiguity of the order of operations performed while evaluating an expressions. Also important for this purpose is the associativity of the operators, associativity defines the direction in which the expression is evaluated when the operator is involved. The precedence and associativity of the operators is summarized below:

C Operator Precedence

Operator(s)	Associativity
( ) [ ] -> .	left to right
! ~ ++ -- - (type) * & sizeof	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
? :	right to left
= += -= etc	right to left
,	left to right

Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, *, /, % all have the same precedence which are higher than that of + and – on the next line.

The sizeof() operator : This is a pseudo-operator given by the language, which returns the number of bytes taken up by a variable or data type. The value returned by this operator can be used to determine the size of a variable.

```
sizeof(int);      // returns 2 on a 16bit machine
sizeof(float);    // returns 4 on a 16bit machine
```

Mixed Mode Expressions and Implicit type Conversions : A mixed mode expression is one in which the operands are not of the same type. In this case, the operands are converted before evaluation, to maintain compatibility of the data types. The following table shows the conversion :

Operand1	**operand 3**	**Result**
char	int	int
int	long	long
int	double	double
int	float	float
int	unsigned	unsigned
long	double	double
double	float	double

e.g.

```
char ch = 'A';
ch = ch + 32;
```

Here, ch will be converted to an integer, since there is an integer(32) in the expression. The result will be an integer.

e.g.

```
float f = 10.0;
int i = 0;
i = f / 3;
```

In this expression , the constant 3 will be converted to a float and then floating point division will take place, resulting in 3.33333, but since the lvalue is an integer the value will be automatically truncated to 3 and the fraction part is lost. (implicit conversion from float to int).

Type Casting : Implicit type conversions, as allowed by the language, can lead to errors creeping in the program if care is not taken. Therefore, explicit type conversions may be used in mixed mode expressions. This is done by type-casting a value of a particular type, into the desired type. This is done by using the (type) operator as follows:

```
(type) expression
```

expression is converted to the given type by the rules stated above.

e.g.

```
float a = 10.0, b = 3.0, c;
c = a / b;
```

This expression will result in 3.333333 being stored in 'c'. If the application requires integer division of two floating point numbers, then the following expression with the type cast can be used:

```
c = (int)a / (int)b;
        or
c = (int) (a / b);
```

Control Flow : The control flow statements are used when it is required that the flow of the program is to be changed after taking some decision. This control flow statement thus specify the order in which the computations are carried out. The language offers *for, while, do-while, if-else, else-if, switch, goto* as control statements. We can classify them as *branching* and *looping structures.*

- **Branching Statements**

  The control flow statements used for branching are if-else, else-if and switch.

  - **The if-else and else-if statements.**

    The syntax of if – else statement is :

```
if(expression 1)
{
        statements;
}
else
{
        statements;
}
```

    If the expression1 is evaluated to be true the statements under it are executed otherwise if it is wrong the statements written after the keyword *else* are executed.

    e.g.

```
if( ch < '0' || ch > '9')
{
        cout << " Not a Number);
}
else                                    // in case of a single statement the braces can be avoided.
        cout<<" This is a number";
```

    A single if(expression) without an else can also be used for checking a single condition.

    A multi-way decision making segment can be written by using if statements inside the else part, which gives the else-if construct.

    e.g.

```
if( option == 1)
        cout<<" 11111";
```

```
else if( option == 2)
        cout<<" 22222";
else if( option == 3)
        cout<<" 33333";
else
        cout << " invalid ";
```

They can be nested as well:

e.g.

```
if(expression 1)
{
    if ( expression 2)
 {
        :
        :
 }
 else
 {

 }
}
else
{
    if ( expression 2)
 {
        :
        :
 }
 else
 {

 }
}
```

- **The Conditional Expression Operator**

An alternate method to using a simple if-else construct is the conditional expressions operator,

?:

A conditional expression operator is a ternary operator, it has three operand, whose general format is:

expression1 ? expression2 : expression3

Here the expression1 is evaluated first, if it is true then the expression 2 is the value of the conditional operator, otherwise the expression 3 is the value.

e.g.

The if else construct

```
if(a>b)
{
        z = a;
}
```

```
else
{
        z = b;
}
```

This can be written as

```
z = (a>b) ? a: b;
```

- **The Switch Construct**

The switch statement is a multi-way decision-making construct that tests whether an expression matches one of a number of constant values, and branches accordingly. The switch statement is another alternative to using nested if-else statements. The general format is as follows :

```
switch(expression)
{

        case value1 : statements;
                                :
                                :
                        break;

        case value2 : statements;
                                :
                                :
                        break;

        case value3 : statements;
                                :
                                :
                        break;

        default     : statements;
                            :
                                                        :
}
```

The expression must be declared in the parentheses, and the body of the switch statement must be enclosed in braces. The values with case should be constants. The expression is evaluated and compared with the values in cases, if it matches the statements under it are executed till it encounters a break statement. If the value does not match with any of the case statements then the statements under the default label are executed.

Omission of break takes control through the next case statements regarded whether the value of case matches or not. This can also be used constructively when same operation has to be performed on a number of cases.

e.g.

```
switch(ch)
{

        case ‘a’ :
        case ‘e’ :
        case ‘i’ :
        case ‘o’ :
```

```
        case 'u' : cout << "Vowel";
                default  : cout << " Consonant ";
    }
```

In the above example if the value of ch is any of a,e,i,o,u then they are printed as vowels , otherwise they are printed as consonants.

- **Looping Statements** : The statements generally used for looping are *for, do-while, while.* The goto *statement can be used for looping, but its use is generally avoided as it leads to haphazard code and also increases the chances of errors.*

  - **The for Loop** : This is a controlled form of loop. The general format is :

```
for( initialize ; test ; update)
{
        statements;
        :
        :
}
```

The initialize part is executed only once, before entering the loop. If test evaluates to false then the next statement after for loop is executed. If the test evaluates to true then the updating takes place.

e.g.

```
for( int i = 0; i< 10; i++) // In C++ you can declare a variable here.
 {
        cout << i;
 }
```

This will print from 0 to 10.

The expressions in the for loop are optional but the semi-colons are necessary, i.e.,

```
for( ; ; )
{
        cout<<"hello";
}
```

This is an infinite loop, with no expression.

e.g.

```
int i = 0
for(; i< 10; )
{
        cout << i;
        i++;
}
```

This is equivalent to our example, which printed from 0 to 9. More than one expression can also be inside the for loop by separating them using commas.

e.g.

```
for( int i = 0, int j=10; i< 10 && y>0 ; i++, y--)
{
        cout << i;
}
```

- **The while Loop**

The for loop is a form of a controlled loop. When the number of iterations to be performed are not known beforehand the while loop can be used. The general format is :

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

The statements in the loop are executed as long as the condition is true .

e.g.

```
int x = 0;
while( x < = 10)
{
        cout << x;
}
```

This example prints from 0 to 10 .

- **The do..while Loop** : The general format is as follows :

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

e.g.

```
do
{
        cout << x;
}while( x < = 10);
```

This works exactly like our previous example . the do.. while loop is exactly like while loop, with one difference. Here testing is done after executing the statements inside the loop and if the condition evaluates to true then we enter the loop again ,whereas in while only after the expression is checked we enter the loop. We can say do loop is executed at least once.

The break statement : The break statement, which was already covered in the switch.. case , can also be used in the loops. When a loop statement is encountered in the loops the control is transferred to the statement outside the loop. When there are nested loops it breaks only from the current loop in which it was written.

The continue statement : The continue statement causes the next iteration of the enclosing loop to begin. When this is encountered in the loop , the rest of the statements in the loop are skipped and control passes to the condition.

Let us see an example that accepts a variable amount of numbers from the keyboard and prints the sum of only positive numbers.

e.g.

```
void main()
{
```

```
        int num, total = 0;
        do
        {
                cout << " enter 0 to quit ";
                cin >> num;                     // equivalent to scanf()
                if(num == 0)
                        break;
                if(num < 0)
                        continue;
                total+=num;
         }while(1);

        cout << total;
}
```

The goto statement : This statement can be used to branch to another statement of the program. This is rarely used as it violates the principle of structured programming. However we can use it when we want to exit from deeply nested loops. The general format is :

```
goto label;
```

where label is a name(tag) followed by a colon.

e.g.

```
for( . . . )
 {
        for( . . . )
        {
                for( . . . )
                 {
                        for( . . . )
                         {
                                if( condition)
                                        goto cleanup;
                         }
                 }
         }
}

cleanup: statements;
        :
        :
```

A good programmer will use control statements effectively in order to avoid the usage of unconditional branching statement – goto.