Module-1

OBJECT ORIENTED PROGRAMMING USING C++

Table of Contents

Object-oriented programming (OOP) is a programming paradigm that represents concepts as "objects" that have data fields(attributes that describe the object) and associated procedures known as methods. Objects, which are instances of classes, are used to interact with one another to design applications and computer programs

An object-oriented program may be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. For example, OOP data structures tend to "carry their own operators around with them" (or at least "inherit" them from a similar object or class) - except when they have to be serialized.

Simple, non-OOP programs may be one "long" list of statements (or commands). More complex programs will often group smaller sections of these statements into functions or subroutines each of which might perform a particular task. With designs of this sort, it is common for some of the program's data to be 'global', i.e. accessible from any part of the program. As programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects.

In contrast, the object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead, the data is accessed by calling specially written functions, commonly called methods, which are either bundled in with the data or inherited from "class objects." These act as the intermediaries for retrieving or modifying the data they control. The programming construct that combines data with a set of methods for accessing and managing those data is called an object. The practice of using subroutines to examine or modify certain kinds of data was also used in non-OOP modular programming, well before the widespread use of object-oriented programming.

An object-oriented program will usually contain different types of objects, each type corresponding to a particular kind of complex data to be managed or perhaps to a real-world object or concept such as a bank account, a hockey player, or a bulldozer. A program might well contain multiple copies of each type of object, one for each of the real-world objects the program is dealing with. For instance, there could be one bank account object for each real-world account at a particular bank. Each copy of

the bank account object would be alike in the methods it offers for manipulating or reading its data., but the data inside each object would differ reflecting the different history of each account.

Objects can be thought of as wrapping their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use. The object's methods will typically include checks and safeguards that are specific to the types of data the object contains. An object can also offer simple-to-use, standardized methods for performing particular operations on its data, while concealing the specifics of how those tasks are accomplished. In this way alterations can be made to the internal structure or methods of an object without requiring that the rest of the program be modified. This approach can also be used to offer standardized methods across different types of objects. As an example, several different types of objects might offer print methods. Each type of object might implement that print method in a different way, reflecting the different kinds of data each contains, but all the different print methods might be called in the same standardized manner from elsewhere in the program. These features become especially useful when more than one programmer is contributing code to a project or when the goal is to reuse code between projects.

Object-oriented programming has roots that can be traced to the 1960s. As hardware and software became increasingly complex, manageability often became a concern. Researchers studied ways to maintain software quality and developed object-oriented programming in part to address common problems by strongly emphasizing discrete, reusable units of programming logic. The technology focuses on data rather than processes, with programs composed of self-sufficient modules ("classes"), each instance of which ("objects") contains all the information needed to manipulate its own data structure ("members"). This is in contrast to the existing modular programming that had been dominant for many years that focused on the *function* of a module, rather than specifically the data, but equally provided for code reuse, and self-sufficient reusable units of programming logic, enabling collaboration through the use of linked modules (subroutines).

A survey by Deborah J. Armstrong of nearly 40 years of computing literature identified a number of fundamental concepts, found in the strong majority of definitions of OOP.[15]

Not all of these concepts are to be found in all object-oriented programming languages. For example, object-oriented programming that uses classes is sometimes called class-based programming, while prototype-based programming does not typically use classes. As a result, a significantly different yet analogous terminology is used to define the concepts of *object* and *instance*.

Benjamin C. Pierce and some other researchers view as futile any attempt to distill OOP to a minimal set of features. He nonetheless identifies fundamental features that support the OOP programming style in most object-oriented languages:[16]

- Dynamic dispatch – when a method is invoked on an object, the object itself determines what code gets executed by looking up the method at run time in a table associated with the object. This feature distinguishes an object from an abstract data type (or module), which has a fixed (static) implementation of the operations for all instances. It is a programming methodology that gives modular component development while at the same time being very efficient.
- Encapsulation (or multi-methods, in which case the state is kept separate)
- Subtype polymorphism
- Object inheritance (or delegation)
- Open recursion – a special variable (syntactically it may be a keyword), usually called this or self, that allows a method body to invoke another method body of the same object. This variable is *late-bound*; it allows a method defined in one class to invoke another method that is defined later, in some subclass thereof.

Similarly, in his 2003 book, *Concepts in programming languages*, John C. Mitchell identifies four main features: dynamic dispatch, abstraction, subtype polymorphism, and inheritance. Michael Lee Scott in *Programming Language Pragmatics* considers only encapsulation, inheritance and dynamic dispatch.

Additional concepts used in object-oriented programming include:

- Classes of objects
- Instances of classes
- Methods which act on the attached objects.
- Message passing
- Abstraction

A Look at Procedure Oriented Programming.

Conventional programming, using high level languages such as COBOL,FORTRAN and C,is commonly known as *procedure oriented programming* (POP). In the POP approach, the problem is viewed as a sequence of things to be done such as reading,calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. A typical structure for procedural programming is shown in the fig 1.1. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

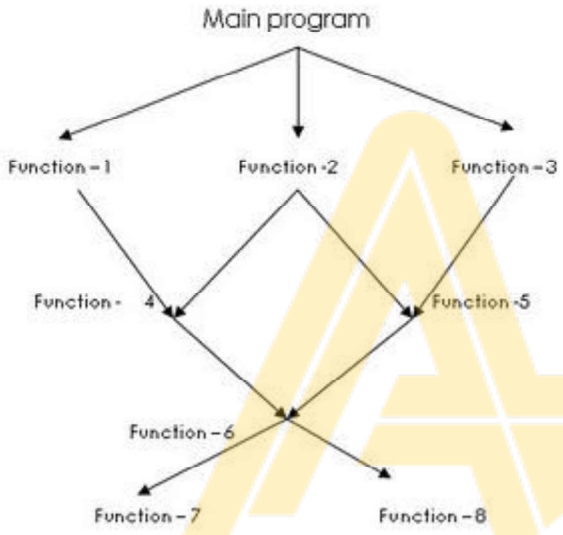


Fig – 1.1 Typical structure of procedure oriented programs

Procedure Oriented programming basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as functions. While concentrating on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them?

DRAWBACKS:

- In a multi-function program, many important data items are placed as*global* so that they may be accessed by all the functions. Each function may have its own *local* data. Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is to be used by which function. In case we need to revise an external data structure, we

also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

Another serious drawback is that it does not model real world problems very well. This is because functions are action oriented and do not really correspond to elements of the problem.

Some characteristics exhibited by procedure-oriented programming are:

1) Emphasis is on doing things (algorithms).
2) Large programs are divided into smaller programs known as functions.
3) Most of the functions share global data.
4) Data move openly around system from function to function.
5) Functions transform data from one form to another.
6) Employs top-down approach in program design.

Object - Oriented Programming Paradigm

The major motivating factor in the invention of object- oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The data of an object can only be accessed by the functions associated with that object. However functions of one object can access the functions of other objects.

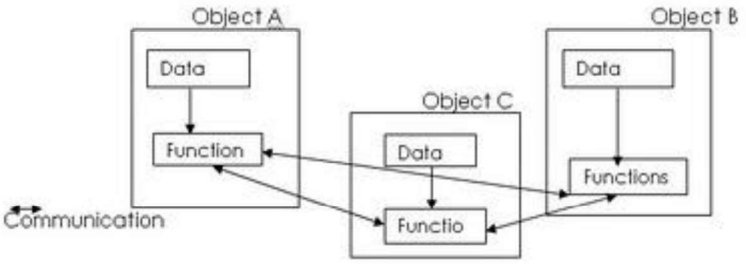


Fig 1.2 The organization of data and functions in object – oriented programs

Some of the striking features of object – oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

Object oriented programming as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

Thus an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since memory partitions are independent, the objects can be used in a variety of different programs without modifications.

Basic Concepts of Object – Oriented Programming

It is necessary to understand some of the concepts used extensively in object – oriented programming. These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, push( ) and pop( ), that can be used for all three stacks. In your program you will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being stored. Thus, the interface to a stack—the functions push( ) and pop( )—are the same no matter which type of stack is being

used. The individual versions of these functions define the specific implementations (methods) for each type of data. Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the specific action (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the general interface. The first objectoriented programming languages were interpreters, so polymorphism was, of course, supported at run time. However, C++ is a compiled language. Therefore, in C++, both run-time and compile-time polymorphism are supported.

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.

When a program is executed, the objects interact by sending messages to each other. For example, If “customer” and “account” are two objects in a program, then the customer object may send a message to account object requesting for the bank balance. Each object contains data and code to manipulate the data. Objects can interact without having to know each others data or code. It is sufficient to know the type of messages accepted, and the type of response returned by the objects.

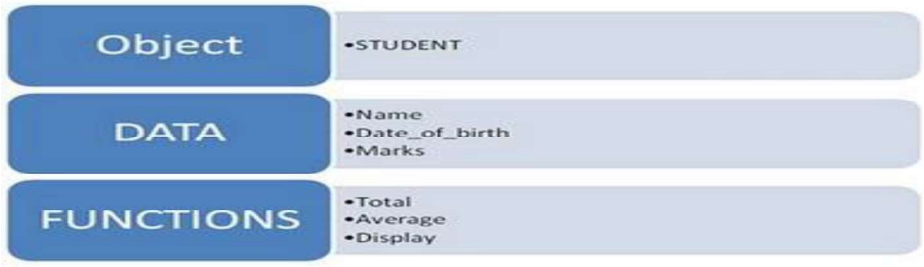


Fig 1.3 Representation of an object

Classes

We just mentioned that objects can 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. Infact 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 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 is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

fruit mango;

will create an object mango belonging to the class fruit.

This is similar to the statement used in C:

int a;

Data Abstraction and 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 outside 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.*

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 members 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*hierarchial classification.* For example, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in fig 1.4.

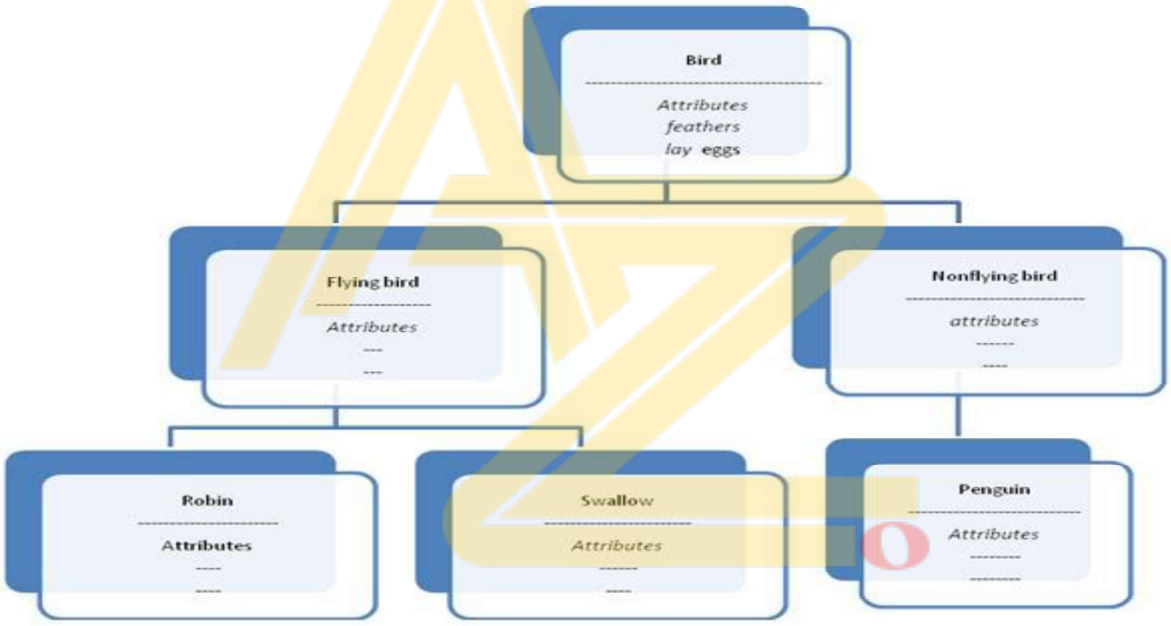


Fig 1.4 Property of Inheritance

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. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Note that each sub-class defines only those features that are unique to it. Without the use of classification, each class would explicitly include all its features.

Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a greek term means 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 a *operator overloading.*

Fig 1.5 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading.*

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

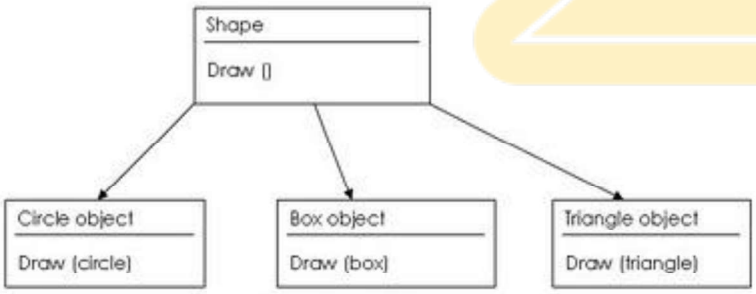


Fig 1.5 Polymorphism

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at rub-time. It is associated with polymorphism and

inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in fig 1.5. By inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

Message passing

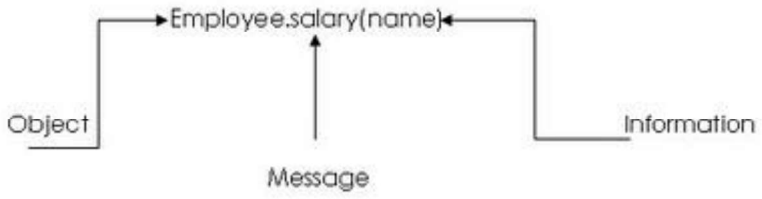
An object- oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language,therefore involves the following basic steps:

1. Creating classes that define objects and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the function (message) and the information to be sent.

Example:



Benefits of OOPs

- Through inheritance, we can eliminate redundant code and extend the use of exiting 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.
- 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 on 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.

A Sample C++ Program

Let's start with the short sample C++ program shown here.

```
#include <iostream>
using namespace std;
int main()
{
int i;
cout << "This is output.\n"; // this is a single line comment
/* you can still use C style comments */
// input a number using >>
cout << "Enter a number: ";
cin >> i;
// now, output a number using <<
cout << i << " squared is " << i*i << "\n";
return 0;
```

Operators

Increment and Decrement

C/C++ includes two useful operators not found in some other computer languages. These are the increment and decrement operators, ++ and −−. The operator ++ adds 1 to its operand, and −−subtracts 1.

In other words:

x = x+1; is the same as ++x; and x = x-1; is the same as x--;

Both the increment and decrement operators may either precede (prefix) or follow (postfix) the operand. For example, x = x+1; can be written

++x; or x++;

There is, however, a difference between the prefix and postfix forms when you use these operators in an expression. When an increment or decrement operator precedes its operand, the increment or decrement operation is performed before obtaining the value of the operand for use in the expression. If the operator follows its operand, the value of the operand is obtained before incrementing or decrementing it.

For instance,

x = 10;

y = ++x; sets y to 11. However, if you write the code as

x = 10;

y = x++;

y is set to 10. Either way, x is set to 11; the difference is in when it happens. Most C/C++ compilers produce very fast, efficient object code for increment and decrement operations— code that is better than that generated by using the equivalent assignment statement. For this reason, you should use the increment and decrement operators when you can.

Here is the precedence of the arithmetic operators:

highest ++ – –

– (unary minus)

lowest + –

Operators on the same level of precedence are evaluated by the compiler from left to right. Of course, you can use parentheses to alter the order of evaluation. C/C++ treats parentheses in the same way as virtually all other computer languages. Parentheses force an operation, or set of operations, to have a higher level of precedence.

The following program contains the function xor( ), which returns the outcome of an exclusive OR operation performed on its two arguments:

```
#include <stdio.h>
int xor(int a, int b);
int main(void)
{
printf("%d", xor(1, 0));
printf("%d", xor(1, 1));
printf("%d", xor(0, 1));
printf("%d", xor(0, 0));
return 0;
   }
```

Variables

A variable is reserved memory location for storing some values.That means to tell the compiler about storage of the variable with help of data-type.

the general syntax is,

```
data-type variable_name = 10;
```

for example,

```
int myvariable = 10;
```

the variable is also referred as an Identifier.

C++ Identifiers

A C++ identifier is a name used for identifying any user defined things.That names refer a variable, function, class, template, or any other custom-defined.Identifiers are sequences of characters

Rules:

- Identifier can be include letters uppercase(A-Z) and lowercase(a-z)
- Identifier can be decimal digits(0-9).
- The first character of an identifier cannot be a digit.
- You Can Include underscore character ?_? in identifiers.The first character of an identifier cannot be an underscore(Some Compilers Accept underscore as a first Character).
- Lowercase letters and uppercase letters are distinct, such that foo and FOO are two different identifiers.
- When using GNU extensions, you can also include the dollar sign character ?$? in identifiers.

Fundamental data types

Data Type	**Description**	**Size**	**Limit**
char	Character or small integer.	1byte signed: -128 to 127	unsigned: 0 to 255
short(intshort)	Short Integer	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value	It can take one of two values: true or false	1byte true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)

Data Type	Description	Size	Limit
double	precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

Declaration Of Variables

In C++, all the variables must be declared before to use or initial statements of the block or main or function or global. variables should specify with data type.
It binds a data type and an identifier with the variable. The data type allows the compiler to execute statements correctly.

```
int a;
float number;
```

simple example program declares the variable, assign value and printing variable.

```
// variable declaration in c++

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables, Its name is b
  int a;

  // assining values in a:
```

```
 a = 5;

 // just printing a value
 cout << a;

 // terminate the program:
 return 0;
}
5
```

Operating Of Variables Example Program

```
// variable declaration and Operating variables in c++

#include <iostream>
using namespace std;

int main ()
{
 // declaring variables , Its name are a,b and c
 int a,b,c;

 // assining values in a:
 a = 5;
 b = 10;

 //Operating variables
 c = a + b;
```

```
  // just printing a result c
  cout << c;

  // terminate the program:
  return 0;
}
15
```

Introduction to strings

1. The C++ language supports strings.
2. Strings data type (string class type) support through the standard string class and Its supports all the basics operations.
3. Strings is not a fundamental type.
4. It behaves in a mostly similar way as basic data types.
5. C++ supports also The C-style character string(Array of Char).

Simple Example Program for C++ string

```
// string variable declaration in c++
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  // declaring variable str for string type
  string str = "Hello C++ String";

  // just printing string
```

```
  cout << str;


  // terminate the program:
  return 0;
}
```

```
Hello C++ String
```

Expression :

A combination of variables, constants and operators that represents a computation forms an expression. Depending upon the type of operands involved in an expression or the result obtained after evaluating expression, there are different categories of an expression. These categories of an expression are discussed here.

• **Constant expressions**: The expressions that comprise only constant values are called constant expressions. Some examples of constant expressions are 20, ' a ' and 2/5+30 .

• **Integral expressions:** The expressions that produce an integer value as output after performing all types of conversions are called **integral expressions.** For example, x, 6*x-y and 10 +int (5.0) are integral expressions. Here, x and yare variables of type into

• **Float expressions:** The expressions that produce floating-point value as output after performing all types of conversions are called **float expressions.** For example, 9.25, x-y and 9+ float (7) are float expressions. Here, x 'and yare variables of type float.

• **Relational or Boolean expressions:** The expressions that produce a bool type value, that is, either true or false are called **relational or Boolean expressions.** For example, x + y<100, m + n==a-b and a>=b + c .are relational expressions.

• **Logical expressions:** The expressions that produce a bool type value after combining two or more relational expressions are called **logical expressions.** For example, x==5 &&m==5 and y>x I I m<=n are logical expressions.

• **Bitwise expressions:** The expressions which manipulate data at bit level are called **bitwise expressions.** For example, a >> 4 and b<< 2 are bitwise expressions.

• **Pointer expressions:** The expressions that give address values as output are called **pointer expressions.** For example, &x, ptr and -ptr are pointer expressions. Here, x is a variable of any type and ptr is a pointer.

• **Special assignment expressions:** An expression can be categorized further depending upon the way the values are assigned to the variables.

• **Chained assignment: Chained assignment** is an assignment expression in which the same value is assigned to more than one variable, using a single statement. For example, consider these statements.

```
a = (b=20); or a=b=20;
```

In these statements, value 20 is assigned to variable b and then to variable a. Note that variables cannot be initialized at the time of declaration using chained assignment. For example, consider these statements.

```
int a=b=30; // illegal

int a=30, int b=30; //valid
```

C++ Operators

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are main keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning. You do not have to memorize all the content of this page. Most details are only provided to serve as a later reference in case you need it.

Arithmetic Operator

They are five arithmetic operators in C++.

- + Addition or unary plus
- - Subtraction or unary minus
- * Multiplication
- / Division
- % Modulo operator

These operators can operate on any arithmetic operations in C++.

Relational Operators

- It Compares two operands and depending on their relationship.

Syntax

```
Op1 Operator Op2
```

Example :

```
a > b
```

There are six relational operators. They are,

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

Logical Operators

- AND, OR operators are used when we want to use two or more Conditions.

Types Of Logical Operators

- && Logical AND
- || Logical OR
- ! Logical NOT

Logical And (&&) Operator

Logical And Operator Definition

If both the operations are successful, then the condition becomes true.

Logical And Operator Syntax

```
expr1 && expr2
```

Logical And Operator Syntax Example

```
if( (a>10) && (a<20) )
  printf(?A is in-between of 10 and 20?);
```

Assignment Operators

They are used to assign the result of an expression to a variable.

Syntax

```
identifier = Expression
```

There are five assignment operators. They are,

```
a=a+1 a+=1

a=a- 1 a-= 1

a=a*2 a*=2

a=a/2 a/= 2

a=a%2 a%=2
```

Advantages:

- Left-hand side operator need not repeat.

- Easy to Read
- More Efficient

Unary Operators

There are two Unary Operators. They are Increment and Decrement.

Increment Unary Operator

```
variable++


++variable;
```

Is Equivalent i=i+1 or i+=1

Increment Unary Operator Types

- Post Increment i++
- Pre Increment ++i

Decrement Unary Operator

```
variable--;


--variable;
```

Is Equivalent i=i-1 or i-=1

Decrement Unary Operator Types

- Post Decrement i--
- Pre Decrement --i

Unary Operators Explanation

- ++i: increments l and then uses its value as the value of the expression;
- i++: uses l as the value of the expression and then increments l;
- --i: decrements l and then uses its value as the value of the expression;
- i--: uses l as the value of the expression and then decrements l.
- Change their original value.

Conditional or Ternary operator

Definition

Check condition if true, it returns first variable value otherwise return second values. sometimes it replaces if..else statement

Syntax

```
Condition? Expression1: Expression2
```

Example

```
(a>10) ? b : c
```

Explanation For Conditional or Ternary operator

Given that

```
a, b, c
```

are expressions;

the expression

```
(a>10) ? b : c
```

has as its value b if a is nonzero, and c otherwise. Only expression b or c is evaluated.

Expressions b and c must be of the same data type. If they are not but are both arithmetic data types, the usual arithmetic conversions are applied to make their types the same. It is also called ternary operators.

The Comma Operator

- The Comma operator can be used to link the related expressions together.

Example For Comma Operator

Comma Operator In for Loops

```
for(i=0,j=1;i>10:i++,j++)
```

Comma Operator In while Loops

```
While(c<10,c--)
```

Scope Resolution Operator

The scope resolution operator is used for the Unary scope operator if a namespace scope (or) Global Scope

Scope Resolution Operator Syntax

```
:: identifier   // for Global Scope
class-name :: identifier  // for Class Scope
namespace :: identifier   // for Namespace Scope


//simple syntax
:: global variable name
```

For more information on Scope Resolution Operator, check this link

new Memory Allocation Operator

Definition

- In C++, new Operator is used to allocating memory at runtime.

Syntax

```
data_type *ptr;
ptr = new data_type[size];
```

delete Memory Releasing Operator

Definition

- In C++, delete operator, is used to release memory or de-allocates memory that was previously allocated by the new operator at runtime or end of the program.

Syntax

```
delete [] ptr;
```

Applications

- The promising areas for application of OOP include:
- Real-time systems
- Simulation and modeling
- Object oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems.

Control Structures in C++

- Conditional structure: if and else
- For Loop
- While Loop
- Do While
- Goto, Break and Continue
- Switch Statement and Break

Conditional structure: if and else

- The if statement executes based test expression inside the braces.
- If statement expression is to true, If body statements are executed and Else body statements are skipped.
- If statement expression is to false If body statements are skipped and Else body statements are executed.
- Simply, Block will execute based on If the condition is true or not.
- IF conditional statement is a feature of this programming language which performs different computations or actions depending on whether a programmer-specified boolean condition evaluates to

true or false. Apart from the case of branch prediction, this is always achieved by selectively altering the control flow based on some condition.

if and else Syntax

```
if (expression) // Body will execute if expression is true or non-zero
{
    //If Body statements
}else
{
    //Else Body statements
}
```

if and else Syntax Example

```
for example, In c
    if (i == 3) {
            doSomething();
    }
    else    {
            doSomethingElse();
    }
```

Syntax Explanation

Consider above example syntax,if (i == 3)

- which means the variable i contains a number that is equal to 3, the statements following the doSomething() block will be executed.
- Otherwise variable contains a number that is not equal to 3, else block doSomethingElse() will be executed.

Example Program For If..else

```
/*  Example Program For If..else In C++ Programming Language */
```

```
// Header Files
#include<iostream>
#include<conio.h>

using namespace std;

//Main Function
int main()
{
    // Variable Declaration
    int a;

    //Get Input Value
    cout<<"Enter the Number :";
    cin>>a;

    //If Condition Check
    if(a > 10)
    {
      // Block For Condition Success
      cout<<a <<" Is Greater than 10";
    }
    else
    {
      // Block For Condition Fail
      cout<<a<<" Is Less than/Equal to 10";
    }
```

```
    // Wait For Output Screen
    getch();

    //Main Function return Statement
    return 0;
}
```

Sample Output:

```
Enter the Number :8
8 Is Less than/Equal to 10

Enter the Number :10
10 Is Less than/Equal to 10
```

Looping and Iteration

This chapter will look at C's mechanisms for controlling looping and iteration. Even though some of these mechanisms may look familiar and indeed will operate in a standard fashion most of the time.

The for statement

The C++ for statement has the following form:

Syntax:

```
for (expression1;Condition;expression2)
    statement;

for (expression1;Condition;expression2) {
    block of statements
}
```

expression1 initialises; expression2 is the terminate test; expression3 is the modifier (which may be more than just simple increment);

NOTE: C/C++ basically treats for statements as while type loops

For loop example program:

```
/*  Example Program For for Loop In C++ Programming Language        */

// Header Files
#include<iostream>
#include<conio.h>

using namespace std;

//Main Function
int main()
{
    // Variable Declaration
   int x=3;

    //for loop
      for (x=3;x>0;x--)
      {
                cout<<"x="<<x<<endl;
      }

    // Wait For Output Screen
    getch();

    //Main Function return Statement
    return 0;
}
```

Output:

```
x=3
x=2
x=1
```

The while statement

The while statement is similar to those used in other languages although more can be done with the expression statement -- a standard feature of C.

The while has the form:

Syntax:

```
while (expression)
    statement;

while (expression)
    block of statements
}
```

While statement example program

```
/*  Example Program For while Loop In C++ Programming Language */

// Header Files
#include<iostream>
#include<conio.h>

using namespace std;

//Main Function
int main()
```

```
{
    // Variable Declaration
   int x=3;


    //while loop
      while (x>0)
      {
                cout<<"x="<<x<<endl;
                x--;
      }

    // Wait For Output Screen
    getch();

    //Main Function return Statement
    return 0;
}
```

Output:

```
x=3
x=2
x=1
```

The do-while statement

Syntax:

C's do-while statement has the form:

```
do {
   statement;
}while (expression);
```

It is similar to PASCAL's repeat ... until except do while expression is true.

do while Loop example:

```
/*  Example Program For Do While Loop In C++ Programming Language */

// Header Files
#include<iostream>
#include<conio.h>

using namespace std;

//Main Function
int main()
{
    // Variable Declaration
   int x=3;

    //do while loop
       do {
         cout<<"x="<<x<<endl;
         x--;
       }while (x>0);

    // Wait For Output Screen
    getch();

    //Main Function return Statement
    return 0;
}
```

outputs:

```
x=3
x=2
x=1
```

NOTE: The postfix x- operator which uses the current value of x while printing and then decrements x.

C/C++ provides two commands to control how we loop: break and continue

```
break -- exit form loop or switch.

continue -- skip 1 iteration of loop.
```

Consider the following example where we read in integer values and process them according to the following conditions. If the value we have read is negative, we wish to print an error message and abandon the loop. If the value read is great than 100, we wish to ignore it and continue to the next value in the data. If the value is zero, we wish to terminate the loop.

Switch Case Statement

In C/C++ programming language, the switch statement is a type of selection mechanism used to allow block code among many alternatives.Simply, It changes the control flow of program execution via multiple blocks.

Switch Statement Rules

- A switch works with the char and int data types.
- It also works with enum types
- Switch expression/variable datatype and case datatype are should be matched.
- A switch block has many numbers of case statements, Each case ends with a colon.
- Each case ends with a break statement. Else all case blocks will execute until a break statement is reached.
- The switch exists When a break statement is reached,
- A switch block has only one number of default case statements, It should end of the switch.
- The default case block executed when none of the cases is true.
- No break is needed in the default case.

Switch Statement Usage

- We can use switch statements alternative for an if..else ladder.
- The switch statement is often faster than nested if...else Ladder.
- Switch statement syntax is well structured and easy to understand.

Switch Statement Syntax:

```
switch ( <expression> or <variable> ) {
      case value1:
       //Block 1 Code Here
       break;

      case value2:
       //Block 1 Code Here
       break;
      ...

      default:
       Code to execute for not match case
       break;
}
```

Example Program For Switch

```
/*  Example Program For Switch Case In C++ Programming Language */

// Header Files
#include<iostream>
#include<conio.h>

using namespace std;
```

```
//Main Function

int main() {
   // Variable Declaration
   char ch;

   //Get Input Value
   cout << "Enter the Vowel (In Capital Letter):";
   cin>>ch;

   //Switch Case Check
   switch (ch) {
      case 'A': cout << "Your Character Is A\n";
         break;

      case 'E': cout << "Your Character Is E\n";
         break;

      case 'I': cout << "Your Character Is I\n";
         break;

      case 'O': cout << "Your Character Is O\n";
         break;

      case 'U': cout << "Your Character Is U\n";
         break;
      default: cout << "Your Character is Not Vowel.Otherwise Not a Capital Letter\n";
         break;
```

```
    }
    // Wait For Output Screen
    getch();


    //Main Function return Statement
    return 0;
}
```

Sample Output:

```
Enter the Vowel (In Capital Letter):A
Your Character Is A


Enter the Vowel (In Capital Letter):O
Your Character Is O



Enter the Vowel (In Capital Letter):h
Your Character is Not Vowel.Or Not a Capital Letter
```

The goto statement

goto allows making an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations. The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:). Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

// goto loop example

```
/*  Example Program For goto In C++ Programming Language */


// Header Files
```

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

using namespace std;

//Main Function

int main() {
   // Variable Declaration
   int num = 10;

   //goto statement declaration
loop:
   cout << num << ", ";
   num--;

   //goto usage
   if (num > 0) goto loop;
   cout << "FIRE!\n";

   // Wait For Output Screen
   getch();

   //Main Function return Statement
   return 0;
}
```

Example programs

1. /* Divide the first number by the second. */

```
#include <stdio.h>
int main(void)
{
int a, b;
printf("Enter two numbers: ");
scanf("%d%d", &a, &b);
if(b) printf("%d\n", a/b);
else printf("Cannot divide by zero.\n");
return 0;
}
```

2. /* Count spaces */

```
#include <stdio.h>
int main(void)
{
char s[80], *str;
int space;
printf("Enter a string: ");
gets(s);
str = s;
for(space=0; *str; str++) {
if(*str != ' ') continue;
space++;
}
printf("%d spaces\n", space);
return 0;
}
```

3. loads a two-dimensional array with the numbers 1 through 12 and prints them row by row.

```
#include <stdio.h>
int main(void)
{
int t, i, num[3][4];
for(t=0; t<3; ++t)
for(i=0; i<4; ++i)
num[t][i] = (t*4)+i+1;
/* now print them out */
for(t=0; t<3; ++t) {
for(i=0; i<4; ++i)
printf("%3d ", num[t][i]);
printf("\n");
}
return 0;
}
```

4. /* A very simple text editor. */

```
#include <stdio.h>
#define MAX 100
#define LEN 80
char text[MAX][LEN];
int main(void)
{
register int t, i, j;
printf("Enter an empty line to quit.\n");
for(t=0; t<MAX; t++) {
printf("%d: ", t);
gets(text[t]);
if(!*text[t]) break; /* quit on blank line */
}
for(i=0; i<t; i++) {
for(j=0; text[i][j]; j++) putchar(text[i][j]);
```

Object Oriented Programming with C++ 10CS36

Dept. of CSE, SJBIT 33

```
putchar('\n');
}
return 0;
}
```

5. Square and square root of a number

```
#include <stdio.h>
int sqr(int x);
int main(void)
{
int t=10;
printf("%d %d", sqr(t), t);
return 0;
}
int sqr(int x)
{
x = x*x;
return(x);
}
```

Reference and Bibliography

1. Object Oriented Programming with C++ E.Balaguruswamy TMH 6th Edition, 2013
2. ObjectOriented Programming with C++ Robert Lafore Galgotia publication 2010
3. ObjectOriented Programming with C++ Sourav Sahay Oxford University 2006
4. Preece, J. Rogers, Y. and Sharp,H., 2007. Interaction Design: Beyond Human Computer Interaction. 2nd ed. New York: John Wiley & Sons, Inc.
5. Preece, J. Rogers, Y. and Sharp,H., 2001. Interaction Design: Beyond Human Computer Interaction. New York: John Wiley & Sons, Inc.
6. Andrew, G and Drew, P, 2009, Use Case Diagrams in Support of Use Case Modeling: Deriving Understanding from the Picture, Journal of Database Management, 20(1), 1- 24, January-March 2009.