Object Oriented Programming

BTCS302-18

What are the Key Ingredients of a Program?

- Common elements in programming languages:
  - Keywords (aka reserved words)
    - Defined in C++. Have a special meaning
  - Programmer-defined entities
    - Names made up by programmer
    - Used to represent variables, functions, etc.
  - Operators
    - Defined in C++
    - For performing arithmetic, comparison, etc. operations
  - Constructs
    - Defined in C++
    - Sequence, selection, repetition, functions, classes, etc.

Every programming language has a syntax (or grammar) that a programmer must observe

The syntax controls the use of the elements of the program

Object Oriented Programming

- C++ is derived from C programming language – an extension that added object-oriented (OO) features.

  - Lots of C++ syntax similar to that found in C, but there are differences and new features.

- Popular because of the reuse of objects
  - Classes – template or blueprint for an object
  - Objects
  - Methods

- Objects are organized in a hierarchy
  - Super Classes
  - Sub Classes

Procedure Oriented Programming

In the Procedure oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions.

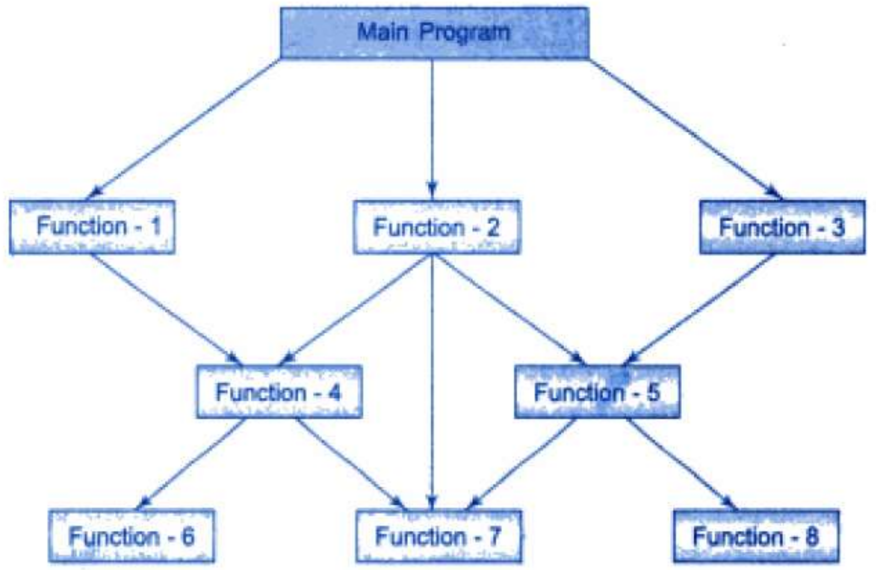


Fig. 1.4 ⇔ *Typical structure of procedure-oriented programs*

Non Procedural/Object Oriented Programming

OOP allows decomposition of a problem into number of entities called objects and then builds data and functions around these objects. The data of an object can be accessed only by the functions associated with that object.

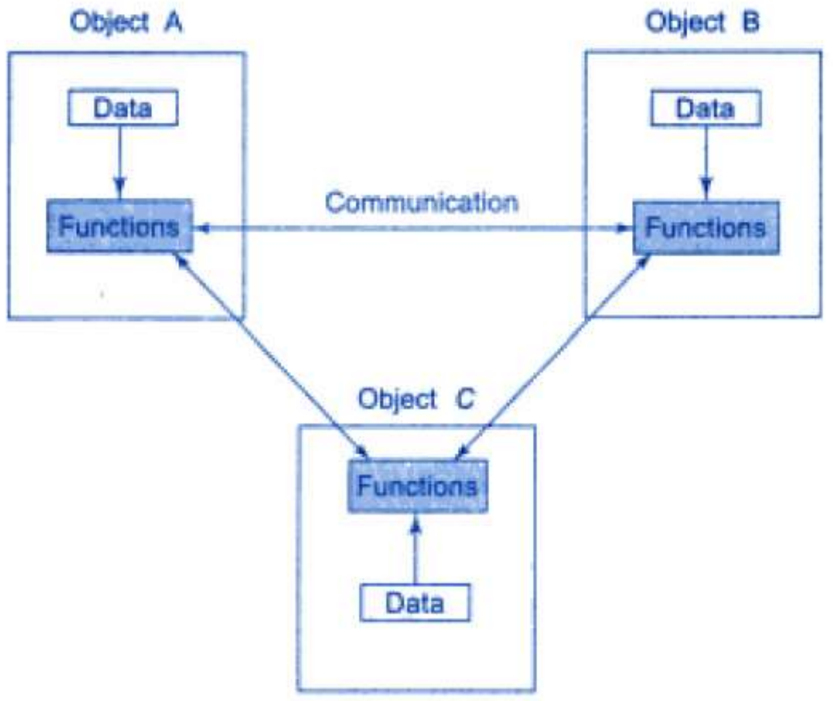
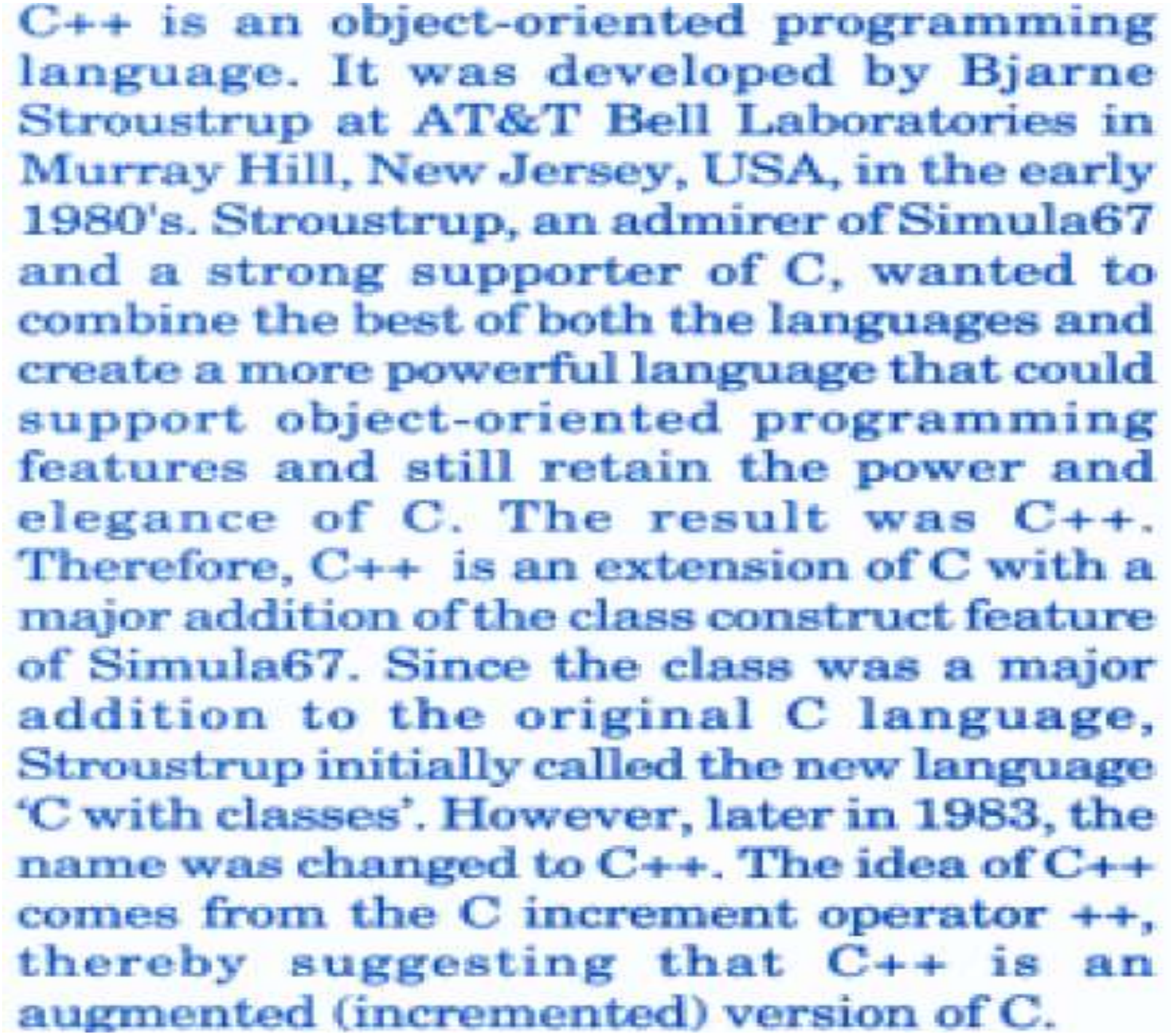


Fig. 1.6 ⇔ *Organization of data and functions in OOP*

Procedural Programming Language	Object Oriented Programming Language
1. Program is divided into functions.	1. Program is divide into classes and objects..
2. The emphasis is on doing things.	2. The emphasis on data.
3. Poor modeling to real world problems.	3. Strong modeling to real world problems.
4. It is not easy to maintain project if it is too complex.	4. It is easy to maintain project even if it is too complex.
5. Provides poor data security.	5. Provides strong data Security.
6. It is not extensible programming language.	6. It is highly extensible programming language.
7. Productivity is low.	7. Productivity is high.
8. Do not provide any support for new data types.	8. Provide support to new Data types.
9. Unit of programming is function.	9. Unit of programming is class.
10. Ex. Pascal , C , Basic , Fortran.	10. Ex. C++ , Java , Oracle.

The Evolution of C++



C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

The simplest C++ program consists of a single function named main.

The syntax of such programs is shown below:

```
#include <iostream>
using namespace std;

int main()
   {
   declaration(s)
   statement(s)
   return 0;
   }
```

The portions of the program shown in blue should always be present. The declarations specify the data that is used by the program. These declarations can declare either constants or variables, for example.

The statements specify the algorithm for the solution to your problem.

Statements

Statements are the executable “units” of a program.

Statements are the translation of an algorithm into program code.

Statements may be

- Simple – typically one line of program code

- Structured – a grouping of a number of statements
  E.g. control structures; functions; etc.

Statements must be terminated with a ; symbol.

Program statements may be executed by one of three control structures:

- Sequence – execute statements one after the other
- Selection – select which statement(s) to execute
- Repetition – repeatedly execute statement(s)

```
#include <iostream>
  using namespace std;

  int main()
     {
     declaration(s)
     statement(s)
     return 0;
     }
```

```
#include <iostream>
   using namespace std;

   int main()
      {
      declaration(s)
      statement(s)
      return 0;
      }
```

The #include Directive

- Preprocessor directive
- Inserts the contents of another file into the program

  iostream is C++ library of input/output functions

  This includes cout and cin
- Do not use ; at end of #include statement

The namespace Directive

This directive allows the cout and cin statements to be used in a program without using the prefix std::

With this directive, may write

cin or cout

Otherwise, we would have to write

std::cout or std::cin

Must use ; at end of namespace directive

```
#include <iostream>
  using namespace std;

  int main()
     {
     declaration(s)
     statement(s)
     return 0;
     }
```

Declarations - Constants and Variables

Constants are data that having unchanging values.

Variables, as their name implies, are data whose values can change during the course of execution of your program.

For both constants and variables, the name, the data type, and value must be specified.

Any variable in your program must be defined before it can be used

Data type specifies whether data is integral, real, character or logical.

Constants

Syntax

```
        const  type  name = expression;
```

The statement must include the reserved word const, which designates the declaration as a constant declaration.

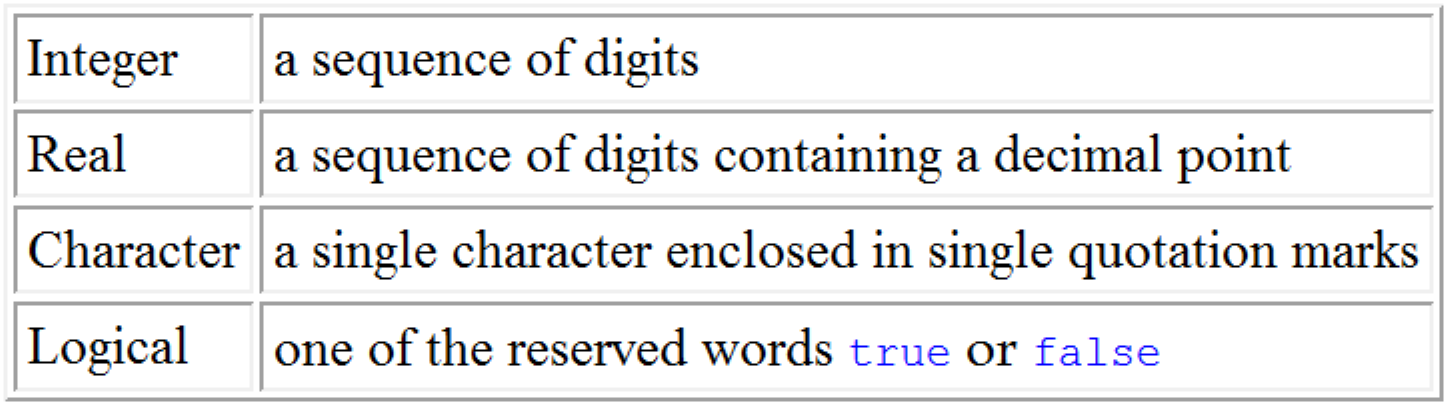
The type specification is optional and will be assumed to be integer.

Examples

```
        const float TAXRATE = 0.0675;
        const int NUMSTATES = 50;
```

Convention is to use uppercase letters for names of constants.

The data type of a constant may be one the types given below



Integer	a sequence of digits
Real	a sequence of digits containing a decimal point
Character	a single character enclosed in single quotation marks
Logical	one of the reserved words `true` or `false`

A Simple, Yet Complete, C++ Program

Program producing output only

```
// Hello world program

#include <iostream>
using namespace std;

int main()
 {
 cout << "Hello world!" << endl;
 return 0;
 }
```

Comment

Preprocessor directives

Function named main() indicates start of the program

Ends execution of main() which ends the program

Let us look at the function main()

```
int main()
    {
    cout << "Hello world!" << endl;
    return 0;
    }
```

This example introduces the notion of a text string

```
        "Hello world!"
```

Integer Data Types

- Designed to hold whole numbers
- Can be signed or unsigned
  - 12 -6 +3
- Available in different sizes (in memory): short, int, and long
- Size of short ≤ size of int ≤ size of long

Floating-Point Data Types

- Designed to hold real numbers

  12.45   -3.8

- Stored in a form similar to scientific notation

- All numbers are signed

- Available in different sizes (space in memory: float, double, and long double

- Size of float ≤ size of double ≤ size of long double

Numeric Types in C++

Number Types

Type	Typical Range
`int`	–2,147,483,648 . . . 2,147,483,647 (about 2 billion)
`unsigned`	0 . . . 4,294,967,295
`short`	–32,768 . . . 32,767
`unsigned short`	0 . . . 65,535
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits

Defining Variables

- Variables of the same type can be defined
  - On separate lines:

```
int length;
int width;
unsigned int area;
```

  - On the same line:

```
int length, width;
unsigned int area;
```

- Variables of different types must be in different definitions

Declaring Variables

```
int x;         // Declare x to be an
            // integer variable;

double radius; // Declare radius to
            // be a double variable;

char a;        // Declare a to be a
            // character variable;
```

Concepts of Object Oriented Programming

Concepts

- Classes
- Objects
- Data Abstraction & Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

Classes

- The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.
- A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

- When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.
- A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows –

```
class Box
{
 public:
double length; // Length of a box
 double breadth; // Breadth of a box
 double height; // Height of a box
 };
```

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Objects

A class provides the blueprints for objects, so basically an object is created from a class. Infact, objects are variable of type class as they represent the real time entities. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box –

```
Box Box1; // Declare Box1 of type Box
Box Box2; // Declare Box2 of type Box
```

Data Abstraction

- Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.
- Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.
- Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

- Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.
- In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

- For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.
- In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this –

```
#include <iostream>
 using namespace std;
 int main()
 {
cout << "Hello C++" <<endl;
 return 0;
 }
```

Benefits of Data Abstraction

- Data abstraction provides two important advantages −
- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.
- By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Encapsulation

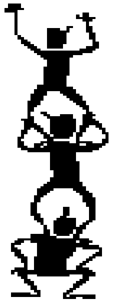
The wrapping up of data and functions into a single unit (called class) is known as encapsulation. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. This insulation of data from direct access by the program is called data hiding or information hiding.

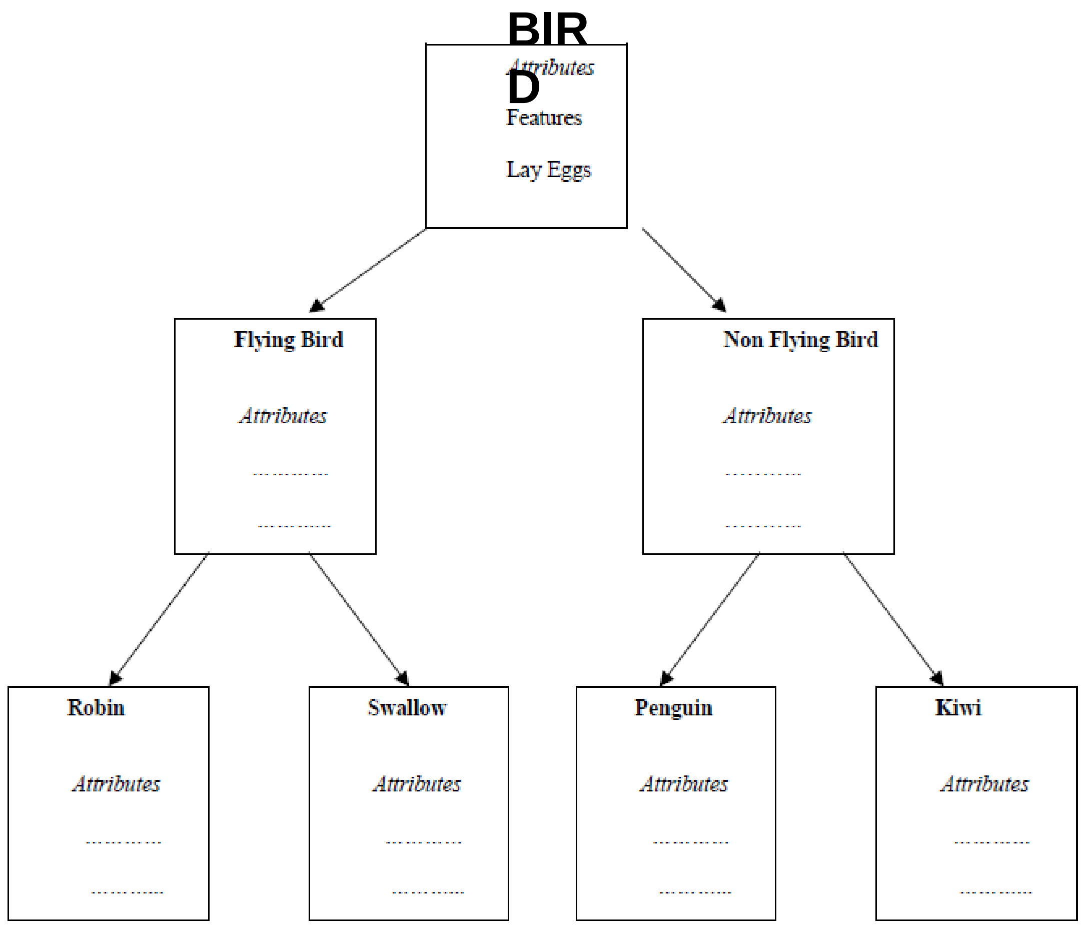
More Concepts of OOP

Abstraction	**Encapsulation**
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
4. **Abstraction**- Outer layout, used in terms of design. For Example:- Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	4. **Encapsulation**- Inner layout, used in terms of implementation. For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.

Inheritance

- *Inheritance* is the process by which objects of one class acquires the properties of objects of another classes.
- It supports the concept of *hierarchical classification.* The principal behind this sort of division is that each derived class s[illegible]s common characteristics with the class [illegible]n which it is derived as illustrated in fig.



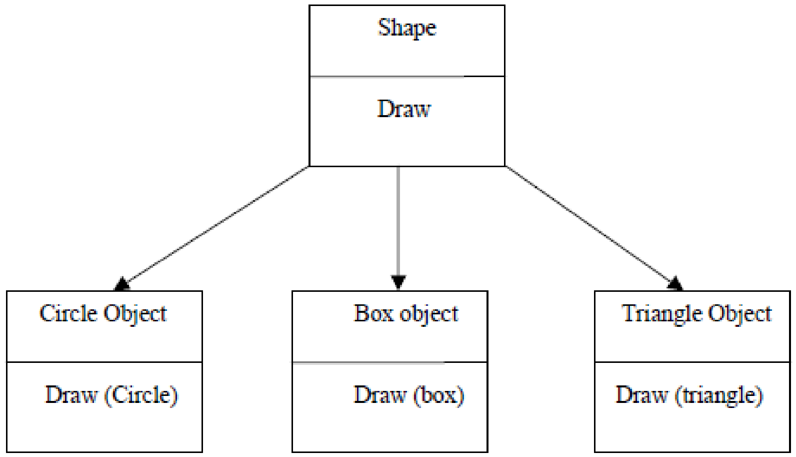

BIRD
Attributes
Features
Lay Eggs
Flying Bird
Attributes
…………
…………
Non Flying Bird
Attributes
…………
…………
Robin
Attributes
…………
…………
Swallow
Attributes
…………
…………
Penguin
Attributes
…………
…………
Kiwi
Attributes
…………
…………

- 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 feature of both the classes.

Polymorphism

- *Polymorphism* is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form.
- An operation may exhibit different behaviour in different instances. The behaviour 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

This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform is known as *function ove*



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 action associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.
- Consider the procedure “draw” in fig. 1.7. 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.

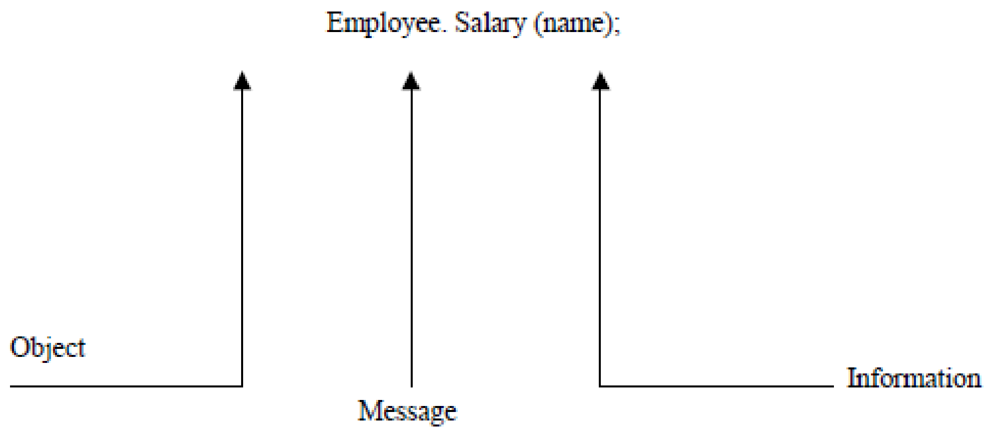
Message Passing

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

1. Creating classes that define object and their behaviour,
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 results. *Message passing* involves specifying the name of object, the name of the function (message) and the information to be sent. Example:


Employee. Salary (name);
Object
Message
Information

Object based & Object Oriented Programming

Object-oriented programming is not the right of any particular languages. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially id designed to support the OOP concepts makes it easier to implement them.

- The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object-based programming languages, and
2. Object-oriented programming languages.

- *Object-based programming* is the style of programming that primarily supports encapsulation and object identity. Major feature that are required for object based programming are:
- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

- Languages that support programming with objects are said to the objects-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

- *Object-oriented programming language* incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statements:

Object-based features + inheritance + dynamic binding

Operators in C++

AVERAGE OF TWO NUMBERS

```
#include<iostream.h> // include header file

Using namespace std;

Int main()

{

          Float number1, number2,sum, average;
          Cin >> number1;         // Read Numbers
          Cin >> number2;         // from keyboard
          Sum = number1 + number2;
          Average = sum/2;
          Cout << "Sum = " << sum << "\n";
          Cout << "Average = " << average << "\n";

          Return 0;

}         //end of example
```

- ***The output would be:***

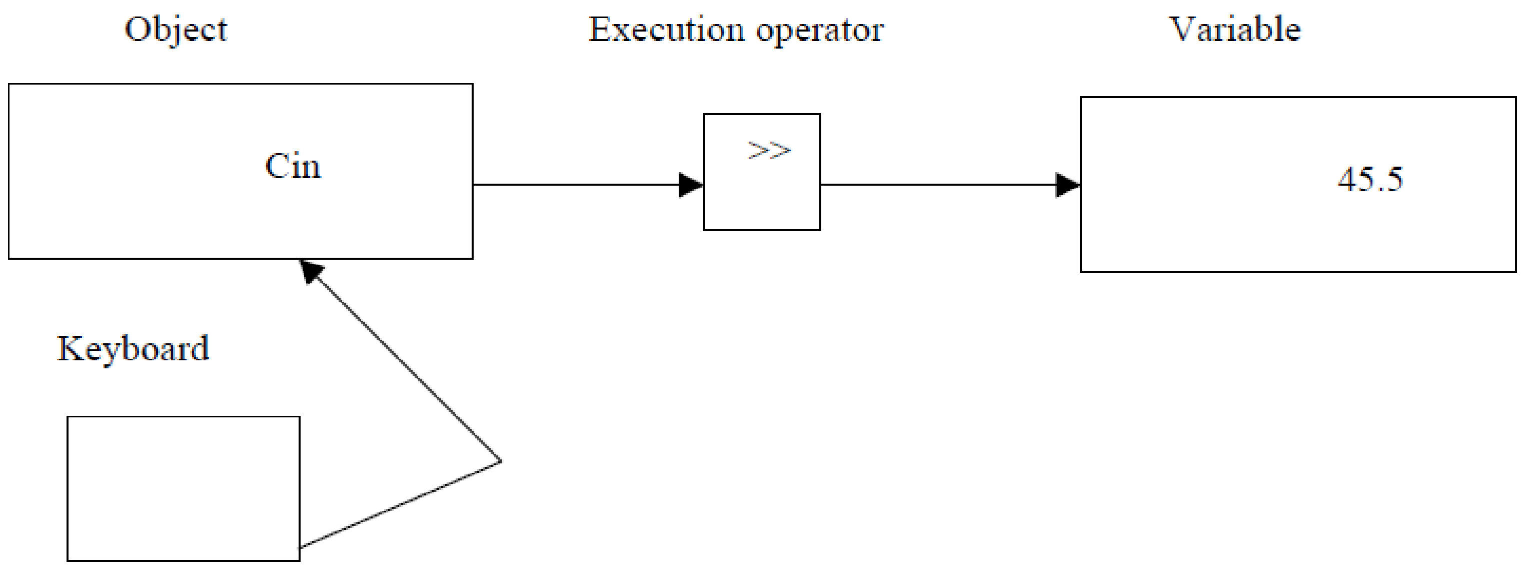
```
Enter two numbers: 6.5 7.5
Sum = 14
Average = 7
```

Input Operator

The Statement

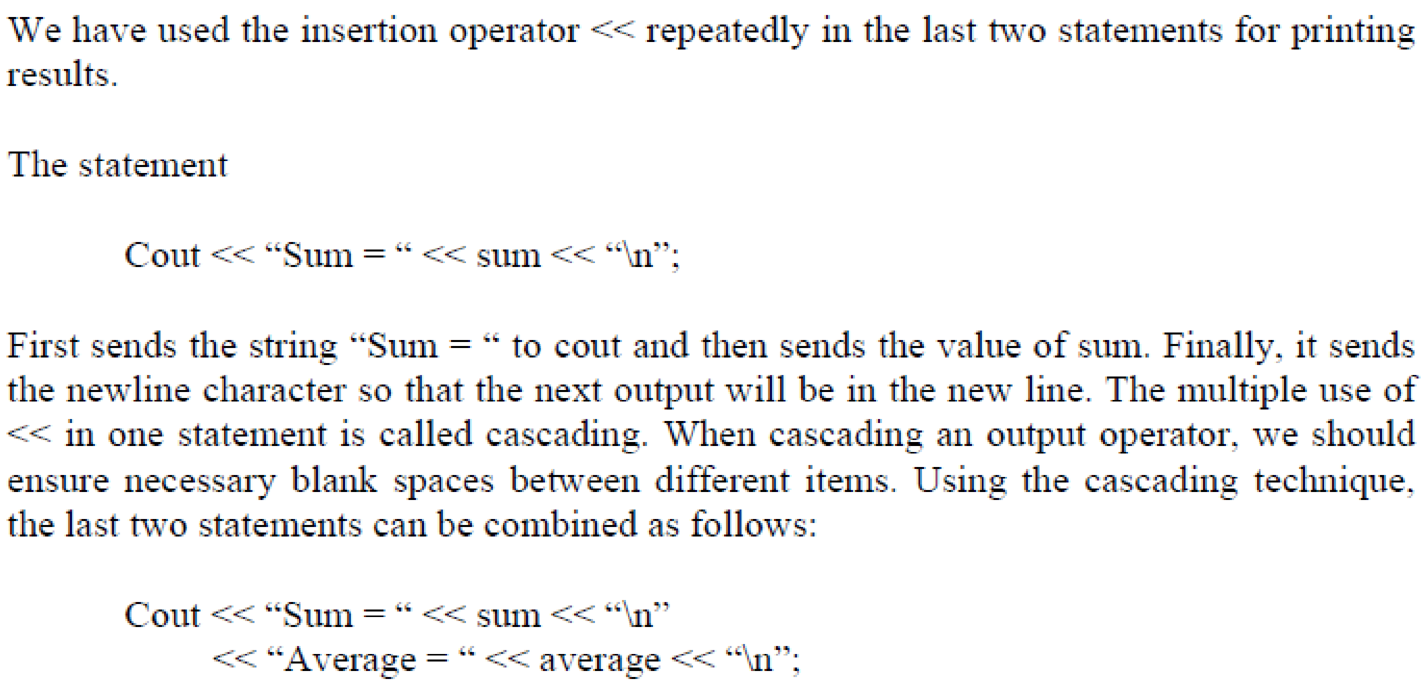
```
cin >> number1;
```

- Is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier cin (pronounced ‘C in’) is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.
- The operator >> is known as extraction or get from operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right fig 1.8. This corresponds to a familiar scanf() operation. Like <<, the operator >> can also be overloaded.



Input using extraction operator

Cascading of I/O Operators



We have used the insertion operator << repeatedly in the last two statements for printing results.

The statement

```
        Cout << “Sum = “ << sum << “\n”;
```

First sends the string “Sum = “ to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called cascading. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
        Cout << “Sum = “ << sum << “\n”
             << “Average = “ << average << “\n”;
```

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

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

The output will be:

```
    Sum  = 14, average = 7
```

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

```
Cin >> number1 >> number2;
```

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to munber1 and 20 to number2.

Scope Resolution Operator

(i) **Class definition.** It describes both data members and member functions.

(ii) **Class method definitions.** It describes how certain class member functions are coded.

We have already seen the class definition syntax as well as an example.

In C++, the member functions can be coded in two ways :

(a) *Inside class definition*

(b) *Outside class definition using scope resolution operator (::)*

Inside Class Definition

- When a member function is defined inside a class, we do not require to place a membership label along with the function name. We use only small functions inside the class definition and such functions are known as **inline** functions.
- In case of inline function the compiler inserts the code of the body of the function at the place where it is invoked (called) and in doing so the program execution is faster but memory penalty is there

Outside Class Definition Using Scope Resolution Operator (::)

In this case the function's full name (qualified_name) is written as shown:

```
Name_of_the_class :: function_name
```

The syntax for a member function definition outside the class definition is :

```
return_type name_of_the_class::function_name (argument list)
  {
    body of function
  }
```

Here the operator::known as scope resolution operator helps in defining the member function outside the class. Earlier the scope resolution operator(::) was used on situations where a global variable exists with the same name as a local variable and it identifies the global variable.

Other Operators

- Unary Operators-The operators that operate a single operand to form an expression are known as unary operators. The operators like + + (increment) operator, -- (decrement) operator etc. are the part of unary operators.
- Binary Operators-The operators that operate two or more operands are known as binary operators. The operators like +, -, *, /, % etc. are binary operators.•
- E.g. a+b, a-b, a*b, a/b etc.
- Ternary Operators-The operator that operates minimum or maximum three operands is known as ternary operator. There is only one ternary operator available in C++. The operator ?: is the only available ternary operator i.e. used as a substitute of if-else statement.
- E.g. a>b ? a:b

Arithmetic Operators

The operators that helps the programmer in mathematical calculations are known as arithmetic operators. Arithmetic operators include (+) for addition, (-) for subtraction, (/) for division, (*) for multiplication etc.

- E.g. 2+5 = 7

Logical Operators

- The operators that help the programmer to connect (combine) two or more expressions, are known as logical operators. Logical operators include:

1. && logical AND
2. | | logical OR
3. | logical NOT

Comparison Operators

The operators that are used to compare variables to check if they are similar or not. It will return values in true or false. These are also known as relational operators. Comparison operators are:

1. > Greater than
2. 2. < Less than
3. 3. = Equal to

Assignment Operators

The operator i.e. used to assign values to identifiers, is known as assignment operator. There is only one assignment operator in C++. The assignment operator (=) is used to assign values to identifiers.

E.g. a = 2 [assign value 2 to a]

Bit-wise Operators

The operators which operate a bit level and allows the programmer to manipulate individual bits. These are basically used for testing or shifting bits.

E.g. x << 3 // Shift three bit position to left

Line Feed Operator & Field Width Operator

These operators are used to format data display.

The most commonly used manipulators are endl and setw.

The endl manipulator has the same effect as using the newline character “n”.

The setw manipulator specifies a field width for printing the value of variables.

Functions

- Functions are the building blocks of C++ programs where all the program activity occurs. Function is a collection of declarations and statements.

Need for a Function

- Monolethic program (a large single list of instructions) becomes difficult to understand. For this reason functions are used. A function has a clearly defined objective (purpose) and a clearly defined interface with other functions in the program. Reduction in program size is another reason for using functions. The functions code is stored in only one place in memory, even though it may be executed as many times as a user needs.

```
//to display general message using function
#include<iostream.h>
include<conio.h>
void main()
{
void disp(); //function prototype
clrscr(); //clears the screen
disp(); //function call
getch(); //freeze the monitor
}
```

```
//function definition
void disp()
{
cout<<“Introduction to Functions\n”;
cout<<”Programming is nothing but logic
implementation”;
}
```

FUNCTION DEFINITION AND DECLARATION

```
Type name_of_the_function (argument list)
{
//body of the function
}
```

- Here, the type specifies the type of the value to be returned by the function. It may be any valid C++ data type. When no type is given, then the compiler returns an integer value from the function.
- Name_of_the_function is a valid C++ identifier (no reserved word allowed) defined by the user and it can be used by other functions for calling this function.

- Argument list is a comma separated list of variables of a function through which the function may receive data or send data when called from other function. When no parameters, the argument list is empty.

```
//function definition add()
void add()
{
int a,b,sum;
cout<<”Enter two integers”<<endl;
cin>>a>>b;
sum=a+b;
cout<<”\nThe sum of two numbers is
“<<sum<<endl;
}
```

The above function add ( ) can also be coded with the help of arguments of parameters as shown below:

```
//function definition add()
void add(int a, int b) //variable names are must in definition
{
int sum;
sum=a+b;
cout<<”\nThe sum of two numbers is “<<sum<<endl;
}
```

DEFAULT ARGUMENTS

C++ allows a function to assign a parameter the default value in case no argument for that parameter is specified in the function call.

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn’t provide a value for the argument with a default value.

```
#include<iostream>
using namespace std;

// A function with default arguments, it can be
called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
        return (x + y + z + w);
}

/* Driver program to test above function*/
int main()
{
        cout << sum(10, 15) << endl;
        cout << sum(10, 15, 25) << endl;
        cout << sum(10, 15, 25, 30) << endl;
        return 0;
}
```

Constant Arguments

In C++, an argument to a function can be declared as const. The argument with constant value should be initialized during the function declaration.

Syntax

type function_name(const data_type variable_name=value);

For Example

int max(const int a=3, int b);   //function prototype

The qualifier const tells the compiler that the value of that argument cannot be modified, and any such attempt will generate a compile time error.

Arguments v/s Parameters

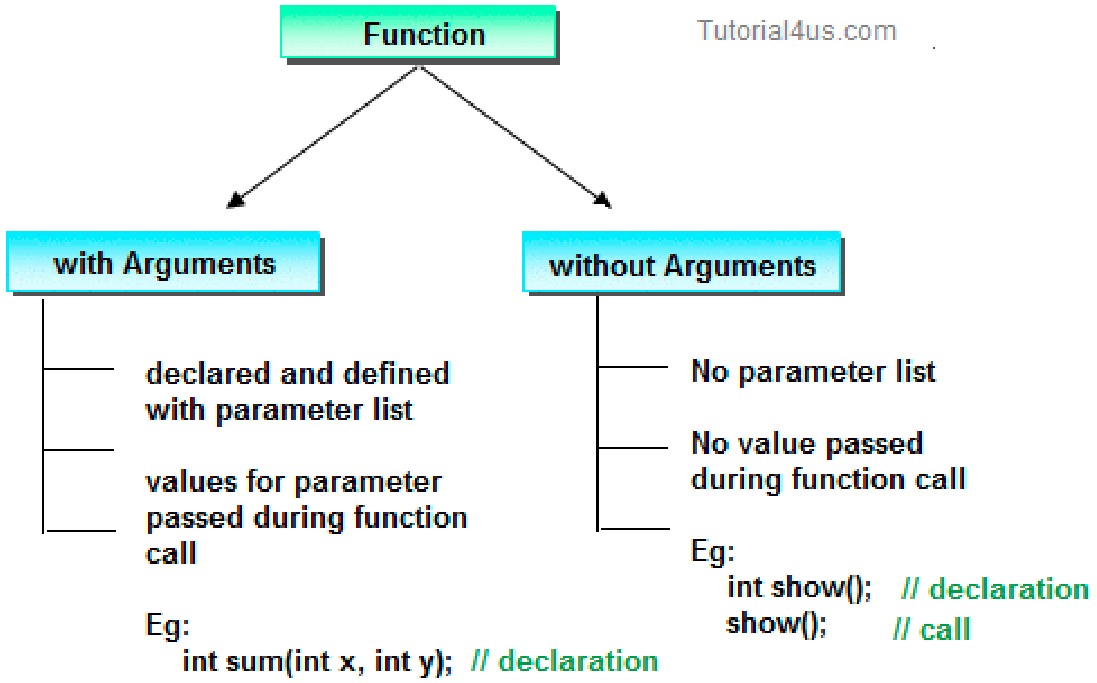
An **argument** is referred to the values that are passed within a function when the function is called. These values are generally the source of the function that require the arguments during the process of execution. These values are assigned to the variables in the definition of the function that is called. The type of the values passed in the function is the same as that of the variables defined in the function definition. These are also called **Actual arguments** or **Actual Parameters**

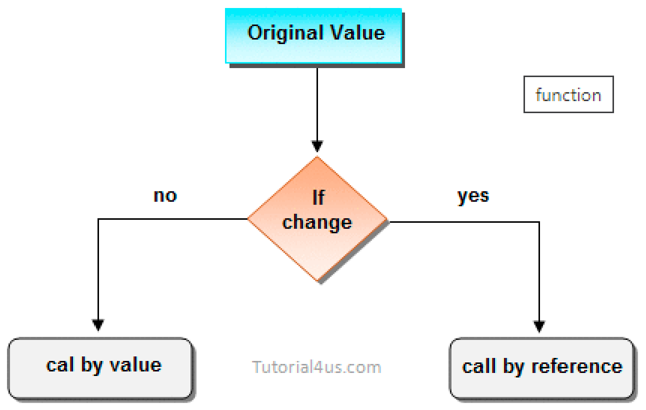
The parameter is referred to as the variables that are defined during a function declaration or definition. These variables are used to receive the arguments that are passed during a function call. These parameters within the function prototype are used during the execution of the function for which it is defined. These are also called Formal arguments or Formal Parameters.

ARGUMENT	PARAMETER
When a function is called, the values that are passed during the call are called as arguments.	The values which are defined at the time of the function prototype or definition of the function are called as parameters.
These are used in function call statement to send value from the calling function to the receiving function.	These are used in function header of the called function to receive the value from the arguments.
During the time of call each argument is always assigned to the parameter in the function definition.	Parameters are local variables which are assigned value of the arguments when the function is called.
They are also called Actual Parameters	They are also called Formal Parameters

Call by Value and Call by Reference in C++

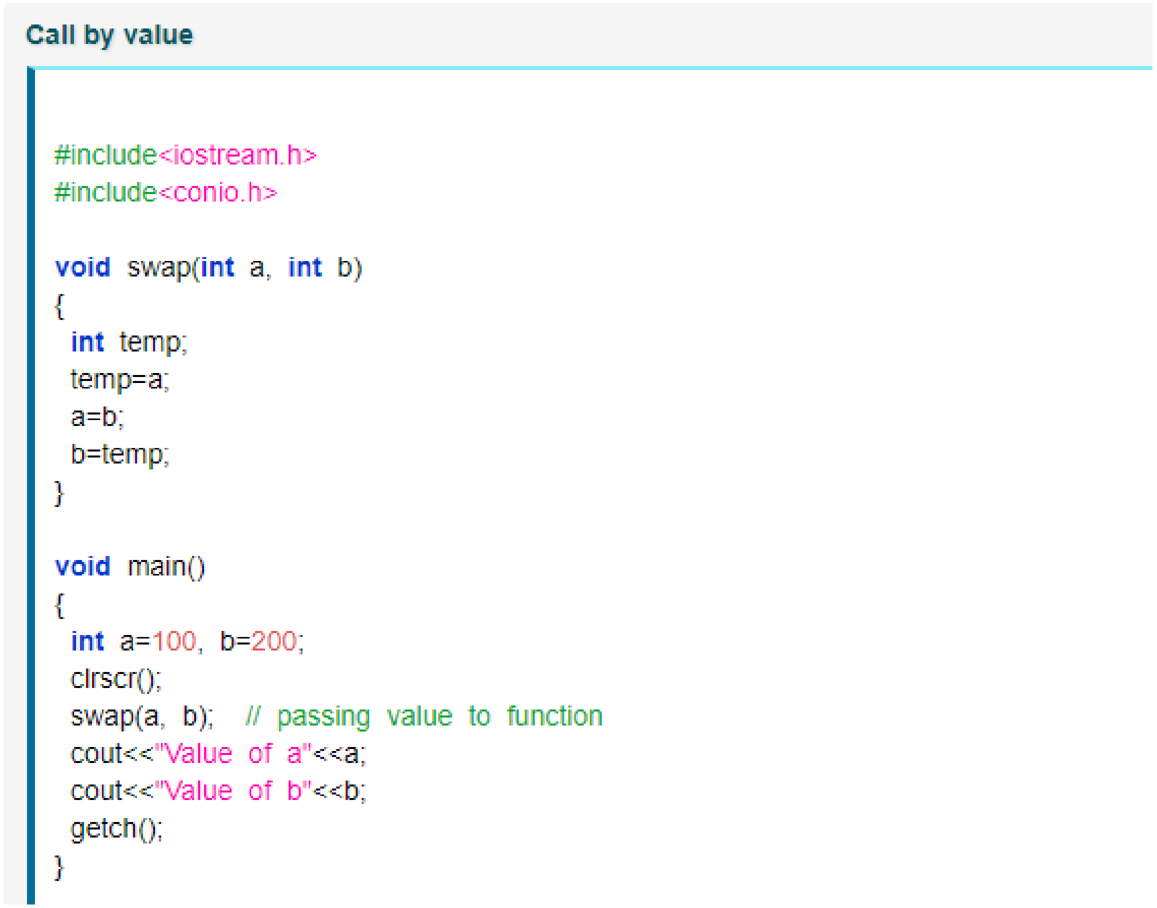
On the basis of arguments there are two types of function are available in C++ language, they are;


Function
Tutorial4us.com
with Arguments
without Arguments
declared and defined with parameter list
values for parameter passed during function call
Eg:
int sum(int x, int y); // declaration
No parameter list
No value passed during function call
Eg:
int show(); // declaration
show(); // call


Original Value
function
If change
no
yes
cal by value
Tutorial4us.com
call by reference

Call by value

In call by value, **original value can not be changed** or modified. In call by value, when you passed value to the function it is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only but it not change the value of variable inside the caller function such as main().



Call by value

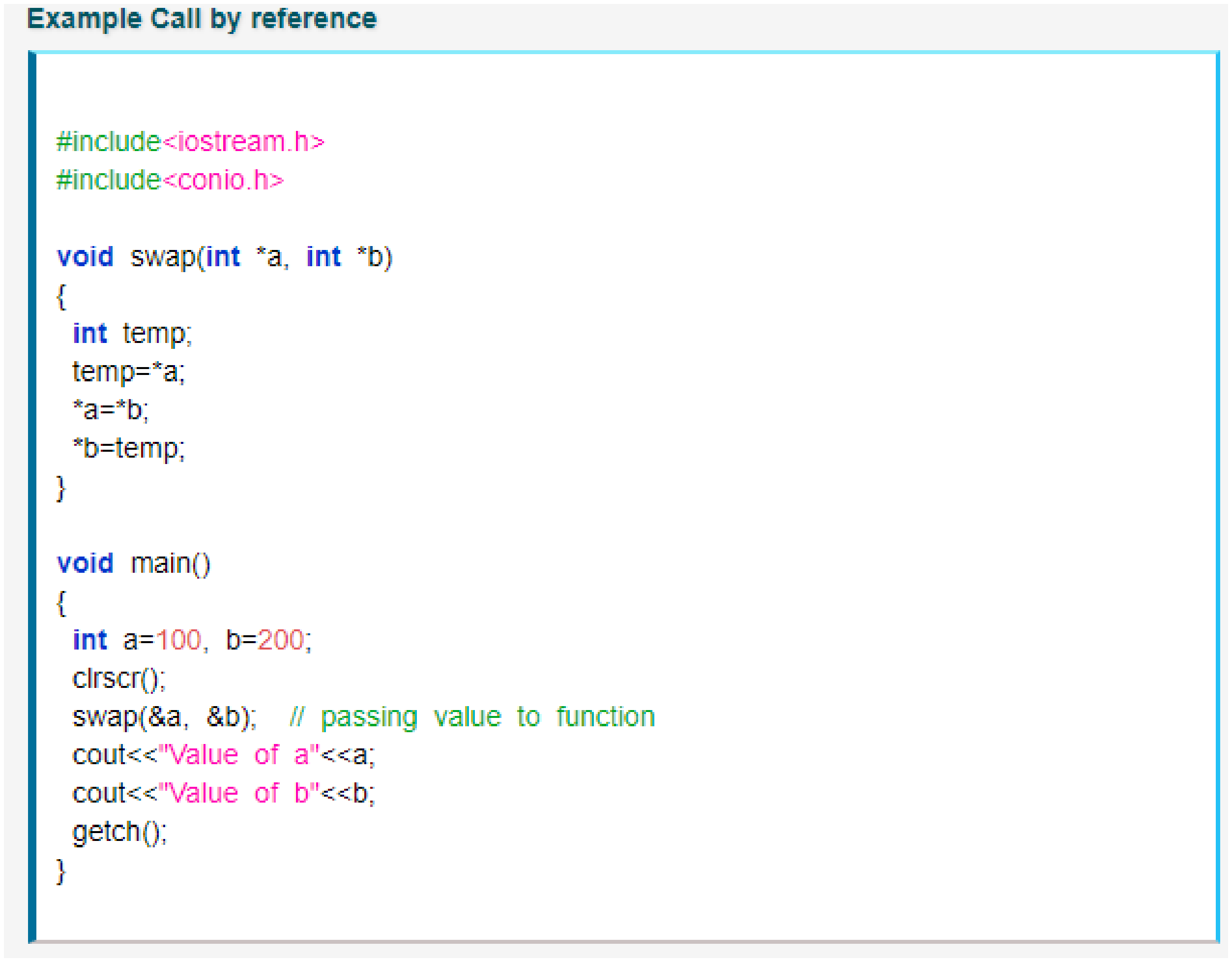
```
#include<iostream.h>
#include<conio.h>

void swap(int a, int b)
{
 int temp;
 temp=a;
 a=b;
 b=temp;
}

void main()
{
 int a=100, b=200;
 clrscr();
 swap(a, b);  // passing value to function
 cout<<"Value of a"<<a;
 cout<<"Value of b"<<b;
 getch();
}
```

Call by reference

In call by reference, **original value is changed** or modified because we pass reference (address). Here, address of the value is passed in the function, so actual and formal arguments shares the same address space. Hence, any value changed inside the function, is reflected inside as well as outside the function.

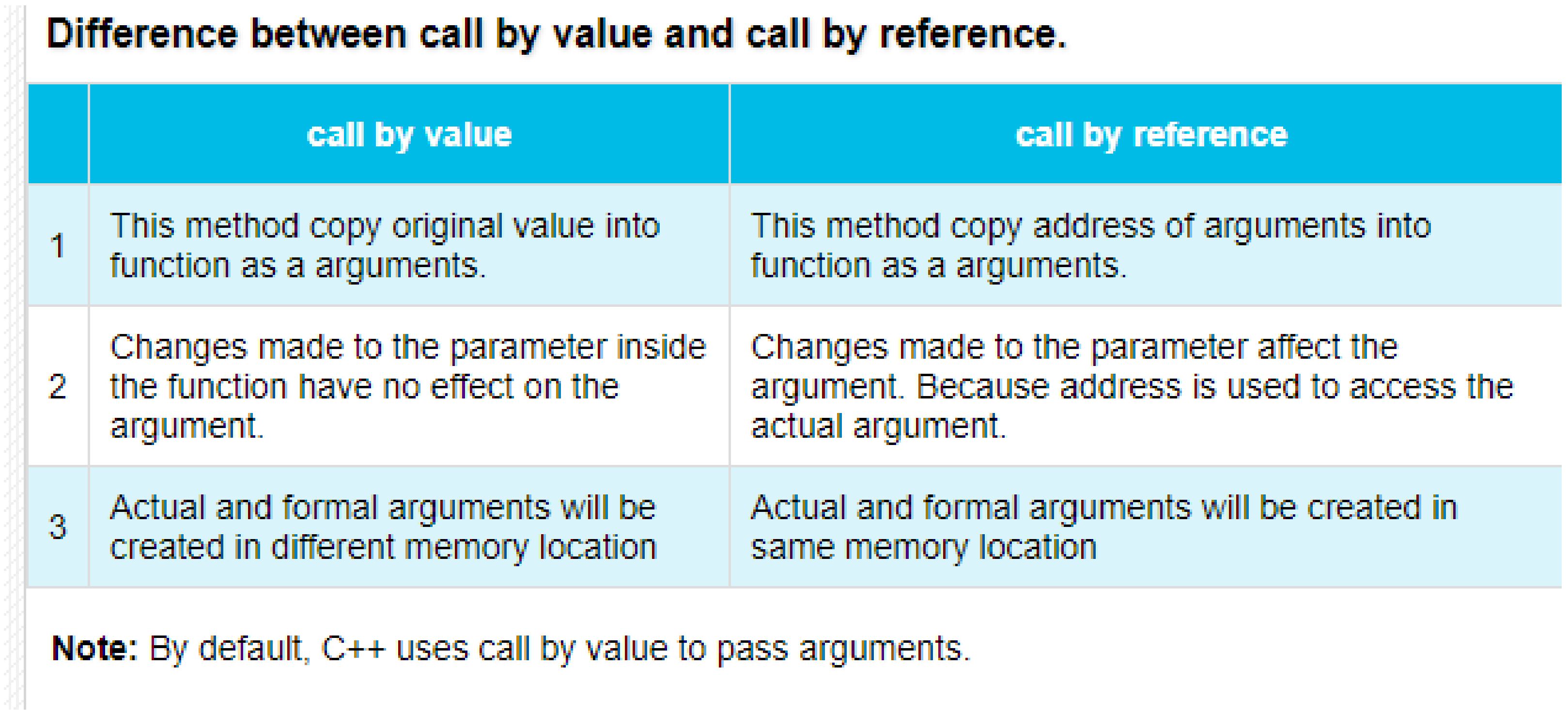


Example Call by reference

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

void  swap(int  *a,  int  *b)
{
  int  temp;
  temp=*a;
  *a=*b;
  *b=temp;
}

void  main()
{
  int  a=100,  b=200;
  clrscr();
  swap(&a,  &b);   //  passing  value  to  function
  cout<<"Value  of  a"<<a;
  cout<<"Value  of  b"<<b;
  getch();
}
```



Difference between call by value and call by reference.

	call by value	call by reference
1	This method copy original value into function as a arguments.	This method copy address of arguments into function as a arguments.
2	Changes made to the parameter inside the function have no effect on the argument.	Changes made to the parameter affect the argument. Because address is used to access the actual argument.
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Note: By default, C++ uses call by value to pass arguments.

Expressions

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types:

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

An expression may also use combinations of the above expressions. Such expressions are known as *compound expressions*.

Constant Expressions

Constant Expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
'x'
```

Integral Expressions

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

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

where **m** and **n** are integer variables.

Float Expressions

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

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

where **x** and **y** are floating-point variables.

Pointer Expressions

Pointer Expressions produce address values. Examples:

```
&m
ptr
ptr + 1
"xyz"
```

where **m** is a variable and **ptr** is a pointer.

Relational Expressions

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

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

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as *Boolean expressions*.

Logical Expressions

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

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

Bitwise Expressions

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

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

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

Special Assignment Expressions

Chained Assignment

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

First 10 is assigned to y and then to x.

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

```
float a = b = 12.34;          // wrong
```

is illegal. This may be written as

```
float a=12.34, b=12.34        // correct
```

Embedded Assignment

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

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+10 = 60 is assigned to x. This statement is identical to

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

Compound Assignment

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

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

may be written as

```
x += 10;
```

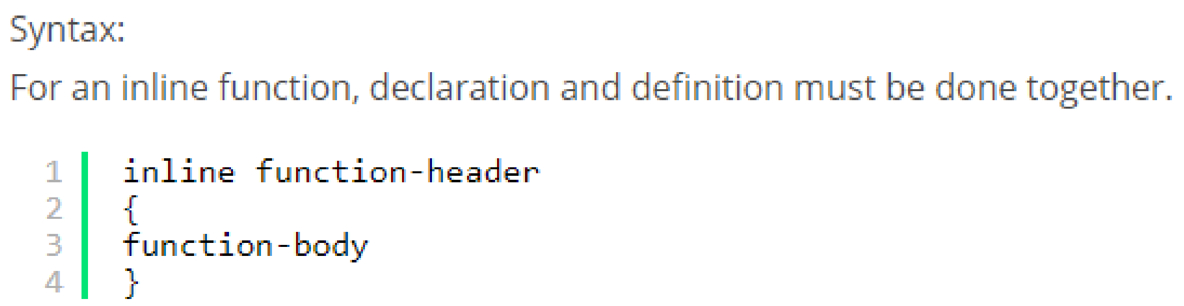
The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is:

Inline Functions

One of the major objectives of using functions in a program is to save memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing tasks such as jumping to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads and sometimes maybe the time taken for jumping to the calling function will be greater than the time taken to execute that

An inline function is a function that is expanded in line when it is invoked thus saving time. The compiler replaces the function call with the corresponding function code that reduces the overhead of function calls.

We should note that inlining is only a request to the compiler, not a command. The compiler can ignore and skip the request for inlining.



Syntax:

For an inline function, declaration and definition must be done together.

```
inline function-header
{
function-body
}
```

Example:

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
return s*s*s;
}
inline int inc(int a)
{
return ++a;
}
int main()
{
int a = 11;
cout << "The cube of 3 is: " << cube(3) << "n";
cout << "Incrementing a " << inc(a) << "n";
return 0;
}
```

When to use Inline function?

We can use Inline function as per our needs. Some useful recommendation are mentioned below-

•We can use the inline function when performance is needed.

•We can use the inline function over macros.

•We prefer to use the inline keyword outside the class with the function definition to hide implementation details of the function.

Advantages of Inline functions

•Function call overhead doesn’t occur.

•It saves the overhead of a return call from a function.

•It saves the overhead of push/pop variables on the stack when the function is called.

•When we use the inline function it may enable the compiler to perform context-specific optimization on the function body, such optimizations are not possible for normal function calls.

•It increases the locality of reference by utilizing the instruction cache.

•An inline function may be useful for embedded systems because inline can yield less code than the function call preamble and return.

Returning Objects

- An object is an instance of a class. Memory is only allocated when an object is created and not when a class is defined. When a function, either a member function or a standalone function, returns an object, we have choices. The function could return
- A reference to an object
- A constant reference to an object

The class Point has two data members i.e. x and y. It has 2 member functions. The function addPoint() adds two Point values and returns an object temp that stores the sum. The function display() prints the values of x and y. The code snippet for this is given as follows.

```
class Point {
   private:
   int x;
   int y;
   public:
   Point(int x1 = 0, int y1 = 0) {
      x = x1;
      y = y1;
   }
   Point addPoint(Point p) {
      Point temp;
      temp.x = x + p.x;
      temp.y = y + p.y;
      return temp;
   }
   void display() {
      cout<<"x = "<< x <<"\n";
      cout<<"y = "<< y <<"\n";
   }
};
```

In the function main(), 3 objects of class Point are created. First values of p1 and p2 are displayed. Then the sum of values in p1 and p2 is found and stored in p3 by calling function addPoint(). Value of p3 is displayed. The code snippet for this is given as follows.

```
Point p1(5,3);
Point p2(12,6);
Point p3;
cout<<"Point 1\n";
p1.display();
cout<<"Point 2\n";
p2.display();
p3 = p1.addPoint(p2);
cout<<"The sum of the two points is:\n";
p3.display();
```