Add the bin subdirectory of your MinGW installation to your **PATH** environment variable so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.

3. BASIC SYNTAX

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what a class, object, methods, and instant variables mean.

- **Object -** Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, and eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.
- **Methods -** A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instant Variables -** Each object has its unique set of instant variables. An object's state is created by the values assigned to these instant variables.

C++ Program Structure:

Let us look at a simple code that would print the words *Hello World*.


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

// main() is where program execution begins.

int main()
{
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

Let us look at the various parts of the above program:

1. The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.
2. The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.

```
add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example:

```
{
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example:

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

is the same as

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

C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in C++.

Here are some examples of acceptable identifiers:

```
mohd       zara    abc    move_name   a_123
myname50   _temp   j      a23b9       retVal
```

C++ Keywords

The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

asm	else	new	this
auto	enum	operator	throw

6. VARIABLE TYPES

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive:

There are following basic types of variable in C++ as explained in last chapter:

Type	Description
bool	Stores either value true or false.
char	Typically a single octet (one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.
wchar_t	A wide character type.

C++ also allows to define various other types of variables, which we will cover in subsequent chapters like **Enumeration, Pointer, Array, Reference, Data structures,** and **Classes**.

Following section will cover how to define, declare and use various types of variables.

Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows:

7. VARIABLE SCOPE

A scope is a region of the program and broadly speaking there are three places, where variables can be declared:

- Inside a function or a block which is called local variables,
- In the definition of function parameters which is called formal parameters.
- Outside of all functions which is called global variables.

We will learn what a function is, and it's parameter in subsequent chapters. Here let us explain what local and global variables are.

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables:

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

int main ()
{
  // local variable declaration:
  int a, b;
  int c;

  // actual initialization
  a = 10;
  b = 20;
  c = a + b;

  cout << c;

  return 0;
}
```

10. STORAGE CLASSES

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

- auto
- register
- static
- extern
- mutable

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
   int mount;
   auto int month;
}
```


The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
   register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

```
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.cpp

```
#include <iostream>

int count ;
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
```

Second File: support.cpp

```
#include <iostream>
```

-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Try the following example to understand all the assignment operators available in C++.

do...while loop	Like a 'while' statement, except that it tests the condition at the end of the loop body.
nested loops	You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

While Loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a while loop in C++ is:

```
while(condition)
{
   statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

One good use of goto is to exit from a deeply nested routine. For example, consider the following code fragment:

```
for(...) {
   for(...) {
      while(...) {
         if(...) goto stop;
         .
         .
         .
      }
   }
}
stop:
cout << "Error in program.\n";
```


Eliminating the **goto** would force a number of additional tests to be performed. A simple**break** statement would not work here, because it would only cause the program to exit from the innermost loop.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form

- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax

The syntax of an if...else if...else statement in C++ is:

```
if(boolean_expression 1)
{
   // Executes when the boolean expression 1 is true
}
else if( boolean_expression 2)
{
   // Executes when the boolean expression 2 is true
}
else if( boolean_expression 3)
{
   // Executes when the boolean expression 3 is true
}
else
{
   // executes when the none of the above condition is true.
}
```


Example

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

int main ()
{
   // local variable declaration:
   int a = 100;

   // check the boolean condition
   if( a == 10 )
```

C++ specifies that at least 256 levels of nesting be allowed for switch statements.

Syntax

The syntax for a **nested switch** statement is as follows:

```
switch(ch1) {
   case 'A':
      cout << "This A is part of outer switch";
      switch(ch2) {
         case 'A':
            cout << "This A is part of inner switch";
            break;
         case 'B': // ...
      }
      break;
   case 'B': // ...
}
```


Example

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

int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;

   switch(a) {
      case 100:
         cout << "This is part of outer switch" << endl;
         switch(b) {
            case 200:
               cout << "This is part of inner switch" << endl;
```

```
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
Call by reference	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used

```
   return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Call by Reference

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.


```
// function definition to swap the values.
void swap(int &x, int &y)
{
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

For now, let us call the function **swap()** by passing values by reference as in the following example:

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

// function declaration
void swap(int &x, int &y);
```

```
Total value is :300
Total value is :120
```

15. NUMBERS

Normally, when we work with Numbers, we use primitive data types such as int, short, long, float and double, etc. The number data types, their possible values and number ranges have been explained while discussing C++ Data Types.

Defining Numbers in C++

You have already defined numbers in various examples given in previous chapters. Here is another consolidated example to define various types of numbers in C++:

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

int main ()
{
   // number definition:
   short  s;
   int    i;
   long   l;
   float  f;
   double d;

   // number assignments;
   s = 10;
   i = 1000;
   l = 1000000;
   f = 230.47;
   d = 30949.374;

   // number printing;
   cout << "short  s :" << s << endl;
   cout << "int    i :" << i << endl;
   cout << "long   l :" << l << endl;
   cout << "float  f :" << f << endl;
```

```
   for ( int j = 0; j < 10; j++ )
   {
      cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
   }

   return 0;
}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result:

```
Element        Value
      0          100
      1          101
      2          102
      3          103
      4          104
      5          105
      6          106
      7          107
      8          108
      9          109
```


Arrays in C++

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer:

Concept	Description
Multi-dimensional arrays	C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Pointer to an array	You can generate a pointer to the first element of an array by simply specifying the array name, without any index.
Passing arrays to functions	You can pass to the function a pointer to an

```
int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   cout << "Greeting message: ";
   cout << greeting << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Greeting message: Hello
```

C++ supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 into the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);**

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

```
using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};
   int *ptr[MAX];

   for (int i = 0; i < MAX; i++)
   {
      ptr[i] = &var[i]; // assign the address of integer.
   }
   for (int i = 0; i < MAX; i++)
   {
      cout << "Value of var[" << i << "] = ";
      cout << *ptr[i] << endl;
   }
   return 0;
}
```


When the above code is compiled and executed, it produces the following result:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows:

```
#include <iostream>

using namespace std;
const int MAX = 4;

int main ()
{
```

```
}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result:

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following:

```
cin >> name >> age;
```

This will be equivalent to the following two statements:

```
cin >> name;
cin >> age;
```

The Standard Error Stream (cerr)

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example.


```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Unable to read....";

   cerr << "Error message : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Error message : Unable to read....
```

The Standard Log Stream (clog)

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Unable to read....";

   clog << "Error message : " << str << endl;
}
```


When the above code is compiled and executed, it produces the following result:

```
Error message : Unable to read....
```

You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs the difference becomes obvious. So it is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

```
   return 0;
}
void printBook( struct Books book )
{
   cout << "Book title : " << book.title <<endl;
   cout << "Book author : " << book.author <<endl;
   cout << "Book subject : " << book.subject <<endl;
   cout << "Book id : " << book.book_id <<endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```


Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&' operator before the structure's name as follows:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

```
   // Print Book1 info, passing address of structure
   printBook( &Book2 );

   return 0;
}
// This function accept pointer to structure as parameter.
void printBook( struct Books *book )
{
   cout << "Book title : " << book->title <<endl;
   cout << "Book author : " << book->author <<endl;
   cout << "Book subject : " << book->subject <<endl;
   cout << "Book id : " << book->book_id <<endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```


The typedef Keyword

There is an easier way to define structs or you could "alias" types you create. For example:

```
typedef struct
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}Books;
```

Now, you can use *Books* directly to define variables of *Books* type without using struct keyword. Following is the example:

```
Books Book1, Book2;
```

You can use **typedef** keyword for non-structs as well as follows:

```
typedef long int *pint32;


pint32 x, y, z;
```

x, y and z are all pointers to long ints.

```
      void setLength( double len );
      double getLength( void );
};

// Member functions definitions
double Line::getLength(void)
{
    return length ;
}

void Line::setLength( double len )
{
    length = len;
}

// Main function for the program
int main( )
{
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length without member function
   line.length = 10.0; // OK: because length is public
   cout << "Length of line : " << line.length <<endl;
   return 0;
}
```


When the above code is compiled and executed, it produces the following result:

```
Length of line : 6
Length of line : 10
```

The private Members

```
   SmallBox box;

   // set box width using member function
   box.setSmallWidth(5.0);
   cout << "Width of box : "<< box.getSmallWidth() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Width of box : 5
```

Constructor & Destructor

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor:


```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // This is the constructor

   private:
      double length;
};

// Member functions definitions including constructor
```

```
   ....
}
```

The Class Destructor

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor:

```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // This is the constructor declaration
      ~Line();  // This is the destructor: declaration

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~Line(void)
{
```

```
    cout << "Normal constructor allocating ptr" << endl;
    // allocate memory for the pointer;
    ptr = new int;
    *ptr = len;
}

Line::Line(const Line &obj)
{
    cout << "Copy constructor allocating ptr." << endl;
    ptr = new int;
   *ptr = *obj.ptr; // copy the value
}

Line::~Line(void)
{
    cout << "Freeing memory!" << endl;
    delete ptr;
}
int Line::getLength( void )
{
    return *ptr;
}

void display(Line obj)
{
   cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program
int main( )
{
   Line line(10);

   display(line);
```

```
Constructor called.
Box2 is equal to or larger than Box1
```

Pointer to C++ Classes

A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator **->** operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

Let us try the following example to understand the concept of pointer to a class:

```
#include <iostream>

using namespace std;

class Box
{
   public:
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume()
      {
         return length * breadth * height;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

```
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members:

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // Constructor definition
```

```
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape
{
   public:
      int getArea()
      {
         return (width * height);
      }
};

int main(void)
{
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```


When the above code is compiled and executed, it produces the following result:

- **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Multiple Inheritance

A C++ class can inherit members from more than one class and here is the extended syntax:

```
class derived-class: access baseA, access baseB....
```

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

```
#include <iostream>

using namespace std;

// Base class Shape
class Shape
{
   public:
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};

// Base class PaintCost
class PaintCost
{
```

25. OVERLOADING (OPERATOR & FUNCTION)

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types:


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

class printData
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }

      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
```

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

   double getVolume(void)
   {
      return length * breadth * height;
   }
   void setLength( double len )
   {
       length = len;
   }

   void setBreadth( double bre )
   {
       breadth = bre;
   }

   void setHeight( double hei )
   {
       height = hei;
   }
   // Overload + operator to add two Box objects.
   Box operator+(const Box& b)
   {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
```

```
      void operator=(const Distance &D )
      {
         feet = D.feet;
         inches = D.inches;
      }
      // method to display distance
      void displayDistance()
      {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }

};
int main()
{
   Distance D1(11, 10), D2(5, 11);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // use assignment operator
   D1 = D2;
   cout << "First Distance :";
   D1.displayDistance();

   return 0;
}
```


When the above code is compiled and executed, it produces the following result:

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

Function Call () Operator Overloading

The function call operator () can be overloaded for objects of class type. When you overload ( ), you are not creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Following example explains how a function call operator () can be overloaded.

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

class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      // overload function call
      Distance operator()(int a, int b, int c)
      {
         Distance D;
         // just put random calculation
         D.feet = a + c + 10;
         D.inches = b + c + 100 ;
         return D;
      }
      // method to display distance
      void displayDistance()
      {
```

```
      if(index >= oc.a.size()) return false;
      if(oc.a[++index] == 0) return false;
      return true;
   }
   bool operator++(int) // Postfix version
   {
      return operator++();
   }
   // overload operator->
   Obj* operator->() const
   {
      if(!oc.a[index])
      {
         cout << "Zero value";
         return (Obj*)0;
      }
      return oc.a[index];
   }
};

int main() {
   const int sz = 10;
   Obj o[sz];
   ObjContainer oc;
   for(int i = 0; i < sz; i++)
   {
       oc.add(&o[i]);
   }
   SmartPointer sp(oc); // Create an iterator
   do {
      sp->f(); // smart pointer call
      sp->g();
   } while(sp++);
   return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result:

```
10
12
11
13
12
14
13
15
14
16
15
17
16
18
17
19
18
20
19
21
```

Designing Strategy

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

28. DATA ENCAPSULATION

All C++ programs are composed of the following two fundamental elements:

- **Program statements (code):** This is the part of a program that performs actions and they are called functions.
- **Program data:** The data is the information of the program which gets affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example:


```
class Box
{
   public:
      double getVolume(void)
      {
         return length * breadth * height;
      }
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions

29. INTERFACES

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
class Box
{
   public:
      // pure virtual function
      virtual double getVolume() = 0;
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```


The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called **concrete classes**.

Abstract Class Example

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()**:

```
#include <iostream>
```

ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows:

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```


Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

34. TEMPLATES

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how they work:

Function Template

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
{
   // body of function
}
```


Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b)
{
    return a < b ? b:a;
}
int main ()
{
```

```
    }
}
```

If we compile and run above code, this would produce the following result:

```
7
hello
Exception: Stack<>::pop(): empty stack
```

Passing Arguments to Threads

This example shows how to pass multiple arguments via a structure. You can pass any data type in a thread callback because it points to void as explained in the following example:

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS     5

struct thread_data{
   int  thread_id;
   char *message;
};

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;

   my_data = (struct thread_data *) threadarg;

   cout << "Thread ID : " << my_data->thread_id ;
   cout << " Message : " << my_data->message << endl;

   pthread_exit(NULL);
}

int main ()
{
   pthread_t threads[NUM_THREADS];
   struct thread_data td[NUM_THREADS];
   int rc;
```

Variable Name	Description
CONTENT_TYPE	The data type of the content, used when the client is sending attached content to the server. For example file upload etc.
CONTENT_LENGTH	The length of the query information that is available only for POST requests.
HTTP_COOKIE	Returns the set cookies in the form of key & value pair.
HTTP_USER_AGENT	The User-Agent request-header field contains information about the user agent originating the request. It is a name of the web browser.
PATH_INFO	The path for the CGI script.
QUERY_STRING	The URL-encoded information that is sent with GET method request.
REMOTE_ADDR	The IP address of the remote host making the request. This can be useful for logging or for authentication purpose.
REMOTE_HOST	The fully qualified name of the host making the request. If this information is not available then REMOTE_ADDR can be used to get IR address.
REQUEST_METHOD	The method used to make the request. The most common methods are GET and POST.
SCRIPT_FILENAME	The full path to the CGI script.
SCRIPT_NAME	The name of the CGI script.
SERVER_NAME	The server's hostname or IP Address.
SERVER_SOFTWARE	The name and version of the software the server is running.

Here is small CGI program to list out all the CGI variables.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

const string ENV[ 24 ] = {
        "COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
        "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
        "HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
        "HTTP_HOST", "HTTP_USER_AGENT", "PATH",
        "QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
        "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
        "SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
        "SERVER_NAME","SERVER_PORT","SERVER_PROTOCOL",
        "SERVER_SIGNATURE","SERVER_SOFTWARE" };

int main ()
{
   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>CGI Environment Variables</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<table border = \"0\" cellspacing = \"2\">";

   for ( int i = 0; i < 24; i++ )
   {
       cout << "<tr><td>" << ENV[ i ] << "</td><td>";
       // attempt to retrieve value of environment variable
       char *value = getenv( ENV[ i ].c_str() );
```

```
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "Radio box selected: " << **fi << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box:

```
<form action="/cgi-bin/cpp_textarea.cgi"
         method="post"
         target="_blank">
<textarea name="textcontent" cols="40" rows="4">
Type your text here...
</textarea>
<input type="submit" value="Submit" />
</form>
```


The result of this code is the following form:

Below is C++ program, which will generate cpp_textarea.cgi script to handle input given by web browser through text area.

```
#include <iostream>
#include <vector>
#include <string>
#include <stdio.h>
```

```
   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Drop Down Box Data to CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("dropdown");
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "Value Selected: " << **fi << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```


Using Cookies in CGI

HTTP protocol is a stateless protocol. But for a commercial website it is required to maintain session information among different pages. For example one user registration ends after completing many pages. But how to maintain user's session information across all the web pages.

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields:

- **Expires:** This showsthe date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.

- The Localization library
- Exception Handling Classes
- Miscellaneous Support Library