C++ as an OOP Language

Building C++ program

- Each class is defined in a separate header (.h) file
- Each class implementation is carried out in a separate .cpp file.
- The program implementation is carried out in a different separate .cpp file.

Preprocessor: #ifdef and #ifndef

- The #ifdef (if defined) and #ifndef (if not defined) preprocessor commands are used to test if a preprocessor variable has been "defined".
- The common use for this is:

  Prevent multiple definitions in header files
- #ifndef checks whether the given token has been #defined earlier in the file or in an included file; if not, it includes the code between it and the closing #else or, if no #else is present, #endif statement. #ifndef is often used to make header files .
- When there definitions in a header file that can not be made twice, the code below should be used. A header file may be included twice other include files include it, or an included file includes it and the source file includes it again.

#ifdef and #ifndef

- To prevent bad effects from a double include, it is common to surround the body in the include file with the following (where MYHEADER_H is replaced by a name that is appropriate for your program).

```
#ifndef MYHEADER_H
#define MYHEADER_H
...
        // This will be seen by the compiler only once
#endif /* MYHEADER_H */
```

Example- Date.h


```
#ifndef DATE_H
#define DATE_H
class Date{
 private :
   int day ,
   month ,
   year ;
 public :
void  init( int iday , int imonth , int iyear) ; // I n i t i a l i z e
void print( ) ; // Pr i n t data
};
#endif
```

Date.cpp


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

 void Date::init( int iday , int imonth , int iyear ){
 // Set
 year = ( iyear >= 0) ? iyear : 0 ;
 month = ( imonth > 0) ? imonth % 13 : 1 ;
 day = ( iday > 0) ? iday % 31 : 1 ;
}
 void Date::print ( ) {
 cout << day << ” . ” << month << ” . ” << year << endl ;
}
```

DateProg.cpp


```
#include "Date.h"
 int main ( ) {
  Date D1 , D2 ;

  // De c l a r e   and   i n i t i a l i z e   o b j e c t s
  D1.init( 1 , 2 , 3 ) ;
  D2.init(-3 ,1 ,-8) ;

  // Pr i n t   o b j e c t s
  D1.print( ) ;
  D2.print( ) ;
  return 0 ;
}
```

The output from this program:

```
1 . 2 . 3
1 . 1 . 0
```

More about classes

Incomplete class declarations

- An incomplete class declaration is a class declaration that does not define any class members.
- You cannot declare any objects of the class type or refer to the members of a class until the declaration is complete.
- However, an incomplete declaration allows you to make specific references to a class prior to its definition as long as the size of the class is not required.
- However, if you declare a class with an empty member list, it is a complete class declaration.

Example


```
class X; // incomplete class declaration
class Z {}; // empty member list
class Y {
public:
  X yobj; // error !!, cannot create an object of an  incomplete class type
  Z zobj; // valid
};
```

Nested classes

- A nested class is declared within the scope of another class.
- The name of a nested class is **local** to its enclosing class.
- Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible construct, including type names, static members, and enumerators from the enclosing class and global variables.
- Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes.
- Member functions of the enclosing class have no special access to members of a nested class.

Example


```
class A {
  int x;
  class B { };
  class C {
      // B b;
      int y;
      void f(A* p, int i) {
       // p->x = i;
       }
  };
  void g(C* p) {
       // int z = p->y;
  }
};
```

- The compiler would not allow the declaration of object b because class A::B is private.
- The compiler would not allow the statement p->x = i because A::x is private.
- The compiler would not allow the statement int z = p->y because C::y is private

friend

- A friend of a class X is a function or class that is not a member of X, but is granted the same access to X as the members of X.
- Functions declared with the friend specifier in a class member list are called friend functions of that class.
- Classes declared with the friend specifier in the member list of another class are called friend classes of that class.
- A class Y must be defined before any member of Y can be declared a friend of another class.

Example 1


```
class X;
class Y {
 public:
  void print(X& x); }; // class Y
class X {
  int a, b;
  friend void Y::print(X& x);
 public:
  X() : a(1), b(2) { } }; // class X
void Y::print(X& x) {
  cout << "a is " << x.a << endl;
  cout << "b is " << x.b << endl;
}
int main() {
  X xobj;
  Y yobj;
  yobj.print(xobj);
}
```

The friend function print is a member of class Y and accesses the private data members a and b of class X.

<u>The following is the output of the above example:</u>
a is 1
b is 2

Example 2


```
#include <iostream>
using namespace std;
class X {
  int a, b;
  friend class F;
public:
  X() : a(1), b(2) { }
};
class F {
public:
  void print(X& x) {
    cout << "a is " << x.a << endl;
    cout << "b is " << x.b << endl;
  }
};
int main() {
  X xobj;
  F fobj;
  fobj.print(xobj);
}
```

- **In this example, the friend class F has a member function print that accesses the private data members a and b of class X and performs the same task as the friend function print in the above example.**
- **Any other members declared in class F also have access to all members of class X.**

The following is the output of the above example:

a is 1

b is 2

Overloading

16

Overload functions

- You overload a function name **f** by declaring more than one function with the name f in the same scope.
- The declarations of **f** must differ from each other by the types and/or the number of arguments in the argument list.
- When you call an overloaded function named **f**, the correct function is selected by comparing the argument list of the function call with the parameter list of each of the overloaded candidate functions with the name **f**.
- A candidate function is a function that can be called based on the context of the call of the overloaded function name.
- You can define many functions with the same name, providing that the return type remains the same and the list of arguments (function signature) changes.
- Overloaded function generally provide related tasks.

Example 1

Example:

```
 void printNumber ( int num ) {
        cout << ” print number as integer : ” << num<< endl ;
}
void printNumber ( char  *str ){
   cout << ” print number as string : ” << atoi (s t r ) << endl ;
}
printNumber ( 1 ) ;
printNumber ( ”7” ) ;
```

Output:

```
print number as integer : 1
print number as string : 7
```

Example2


```
#include <iostream>
using namespace std;
void print(int i) {
  cout << " Here is int " << i
  << endl;
}
void print(double f) {
  cout<< " Here is double
  "<<f<< endl;
}
void print(char* c) {
  cout<<" Here is char*
  "<<c<<endl;
}
```

```
int main() {
  print(10);
  print(10.10);
  print("ten");
}
```

The following is the output of the above example:

```
Here is int 10
Here is double 10.1
Here is char* ten
```

Overload operator

- To overload an operator in order to use it with classes we declare *operator functions*, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload.

- Overloading an operator is similar to defining a function, with the use of the operator keyword.

- An operator can be overloaded as a member function, or outside the class it servers as a friend function.

- The format is:

      type operator sign (parameters) { /*...*/ }

Operators to overloading

+ - * / = < > += -= *= /= << >>
<<= >>= == != <= >= ++ -- % & ^ ! |
~ &= ^= |= && || %= [] () , ->* -> new
delete new[] delete[]

Restrictions for overload operators

- Operators that cannot be overloaded: ”.” ”.*” ”::” ”?:” ”sizeof”
- The precedence of operators cannot be changed; overloaded ”*” will always evaluate before overloaded ”+”.
- The number of arguments that an operator takes cannot be changed: ”+” will always have two operands.
- Each operator have to be overloaded explicitly: Overloading ”+” does not mean an automatic overloading of the different operator ”+=”.

Example


```
// vectors: overloading
   operators
 //  example
#include <iostream>
using namespace std;
class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator +
   (CVector);
};

CVector::CVector (int a, int b)
   {
  x = a;
  y = b;
}
```

```
CVector CVector::operator+
  (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}

 int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}
```

Output:

4,3

Example


```
class X { };
void operator!(X) {
  cout << "void operator!(X)" << endl;
}
class Y {
 void operator!() {
     cout << "void Y::operator!()" << endl;
  }
};
class Z { };
 int main() {
X ox;
Y oy;
Z oz;
!ox;
!oy;
!oz; <--incorrect
 }
```

The pointer this

- The keyword **this** identifies a special type of pointer.
- Suppose that you create an object named x of class A, and class A has a nonstatic member function f(). If you call the function x.f(), the keyword this in the body of f() stores the address of x. You cannot declare the **this** pointer or make assignments to it.
- A static member function does not have a **this** pointer.
- The type of the **this** pointer for a member function of a class type X, is X* const. If the member function is declared with the const qualifier, the type of the **this** pointer for that member function for class X, is const X* const.
- A const **this** pointer can by used only with const member functions. Data members of the class will be constant within that function.
- The function is still able to change the value, but requires a const cast to do so.

Example


```
#include <iostream>
using namespace std;
class X {
  int a;
public:
  void Set_a(int a) {
     // The 'this' pointer is used to retrieve 'xobj.a'
     // hidden by the automatic variable 'a'
       this->a = a;
  }
     void Print_a() { cout << "a = " << a << endl; }
};
int main() {
  X xobj;
  int a = 5;
  xobj.Set_a(a);
  xobj.Print_a();
}
```

In the member function Set_a(), the statement this->a = a uses the this pointer to retrieve xobj.a hidden by the automatic variable a.

Output : 5

Constructors

Overview on constructors

- Constructors and destructors are member functions which have no return value, not even void.
- Constructors may receive parameters but destructors receive no parameters.
- Constructors and destructors do not have return types nor can they return values.
- Constructors are called upon initialization of an object and destructors are called when an object is deleted.
- A constructor name must be the same as its class name.
- A destructor's name is ~<class name>
- Constructors cannot be declared as static or const.
- Otherwise constructors behave the same as regular functions. They can be overloaded, or have default parameter values.
- When not specifically declared by the user, the C++ compiler assigns a default constructor ().
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.

Example


```
#include <cstring>
class Y {
private:
  char * str;
  int number;
public:
  Y(const char*, int); // Constructor
  ~Y() {
    delete[] str; }    // Destructor
};
// Define class Y constructor
Y::Y(const char* n, int a) {
   str = strcpy(new char[strlen(n) + 1 ], n);
   number = a;
}

int main () {
 // Create and initialize object of class Y
 Y yobj = Y("somestring", 10);
 // ...
 // Destructor ~Y is called before  // control returns from main()
}
```

Copy constructors

- The copy constructor lets you create a new object from an existing one by initialization.
- A copy constructor of a class A is a non-template constructor in which the first parameter is of type A&, const A&, volatile A&, or const volatile A&, and the rest of its parameters (if there are any) have default values.
- If you do not declare a copy constructor for a class A, the compiler will implicitly declare one for you, which will be an inline public member.

Example


```
#include <iostream>
using namespace std;
 class A {
  int i;
  A() : i(10) { }
};
class B {
  int j;
  B() : j(20) {
    cout << "Constructor B(), j = " << j << endl;
  }
  B(B& arg) :j(arg.j) {
      cout << "Copy constructor B(B&), j = " << j << endl;
  }
  B(const B&, int val = 30) : j(val) {
      cout << "Copy constructor B(const B&, int), j = " << j << endl;
  }
};
 class C {
  C() { }
  C(C&) { }
};
```

Example (cont.)


```
int main() {
  A a;
  A a1(a);
  B b;
  const B b_const;
  B b1(b);
  B b2(b_const);
  const C c_const;
//  C c1(c_const);
}
```

The following is the output of the above example:

Constructor B(), j = 20
Constructor B(), j = 20
Copy constructor B(B&), j = 20
Copy constructor B(const B&, int), j = 30

- **The statement A a1(a) creates a new object from a with an implicitly defined copy constructor.**
- **The statement B b1(b) creates a new object from b with the user-defined copy constructor B::B(B&).**
- **The statement B b2(b_const) creates a new object with the copy constructor B::B(const B&, int).**
- **The compiler would not allow the statement C c1(c_const) because a copy constructor that takes as its first parameter an object of type const C& has not been defined.**