OOP with C++

Lecture 11-1

Contents

OOP with C++

- C++ provides language constructs to build a program in an object-oriented style.
- Similar to Java, C++ uses classes and objects as the two main aspects of object-oriented programming.
- A class is a template for objects, and an object is an instance of a class.
  - When the individual objects are created, they inherit all the variables and functions from the class.

Class Declaration

- As in Java, class is the fundamental unit to enable object-oriented programming.
  - They can contain attributes and functions
- An object is an instantiation of a class.

```
class class_name {
 access_specifier :
   member1;
 access_specifier:
   member2;
 ...
};
```

```
class Rectangle {
    int width, height;
 public:
    void set_values (int,int);
    int area (void);
};
```

```
// Create an object
Rectangle rect;
```

Data Members

- A variable declared in a class is called a data member. It can be accessed with dot operator(.).

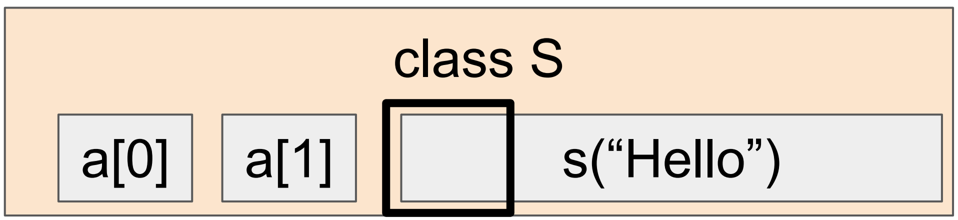
```cpp
#include <iostream>
#include <string>
using namespace std;

 class S {
    public:
       int n;
       int a[2] = {1, 2};
       string s = "Hello";
 };
```

```cpp
int main(){
   S s;
   cout << s.s << endl;
   Cout << s.a[0] << ","
         << s.a[1] <<endl;
}
```

```
Hello
1,2
```

Data Members

- C++ will not throw an error even if the array index is out of bounds, when there is a subsequent data member defined after the array definition.



a[2] access attempt returns data here

```cpp
#include <iostream>
#include <string>
class S {
   public:
      int a[2] = {1, 2};
      string s = "Hello";
};
```

```cpp
int main(){
 S s;
 cout << s.a[0] << ","
      << s.a[1] << endl;
 cout << s.a[2] << endl;
}
```

```
1,2
6487808
```

Member Functions

- Functions declared inside the class are called member functions.
- It can be accessed with dot operator(.).

```
class MyClass {
   public:
       void myMethod() {
              cout<<"Hello!";
           }
};
```

```
int main() {
  MyClass myObj;
  myObj.myMethod();
  return 0;
}
```

```
Hello!
```

Function Definition with Scope Operator (::)

- Function declaration is done inside the class, but it can be defined outside the class with (::).

```
#include <iostream>
using namespace std;
class Rectangle {
   int width, height;
 public:
   void set_values (int,int);
   int area () {return width * height;}
};
void Rectangle::set_values(int x, int y) {
   width = x;
   height = y;
}
```

```
Rectangle r;
r.set_values(5,6);
cout<<rect.area();
return 0;
```

```
30
```

Object Instantiation: Variable

- Objects can be declared as variables (without new).
- Members of an object can be accessed using ".".

```
#include <iostream>
using namespace std;
class Rectangle {
      int width, height;
   public:
      Rectangle(int x, int y) : width(x), height(y) {}
      int area () { return width*height; }
};
```

```
int main() {
   Rectangle obj(3,4);
   cout << "obj's area: " << obj.area() << '\n';
   return 0;
}
```

Object Instantiation: Pointer

- Objects can also be pointed to by pointers.
  - It can be instantiated by the keyword new.
  - It must be deallocated with the explicit call of delete.
- Members can be accessed with “->” operator.

```cpp
int main() {
   Rectangle obj(3,4);
   Rectangle* foo;
   foo = new Rectangle(5, 6);
   cout << "obj's area: "
        << obj.area() << '\n';
   cout << "*foo's area: "
        << foo->area() << '\n';
   delete foo;
   return 0;
}
```

```
obj's area: 12
*foo's area: 30
```

this pointer

- The keyword *this* is a pointer whose value is the address of the instance.
- Equivalent to Java *this*, except it is a pointer.

```
class Test {
   private:
      int x;
   public:
      void setX (int x) {
         this->x = x;
      }
      void print() {
         cout << "x = " << x ;
      }
};
```

```
int main() {
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
  return 0;
}
```

```
x = 20
```

Forward Declaration

- Separates the declaration and definition of a class.
- Other classes defined before the class can use it.

```
class A; // Class Declaration
class B {
    int x;
 public:
    void setdata(int n){ x = n;}
    int sum(A, B);
};
class A { // Class Definition
    int y;
 public:
    void setdata(int m){y = m;}
};
int B::sum(A m, B n) {
    return m.y + n.x;
}
```

```
int main(){
   B b; b.setdata(4);
   A a; a.setdata(5);
   cout << "The sum is : "
        << b.sum(a, b);
   return 0;
}
```

```
The sum is : 9
```

Special Member Functions

- There are member functions that use a special syntax for their declarations
  - Constructor
  - Default constructor
  - Copy constructor
  - Copy assignment operator
  - Destructor
  - And more (e.g., move constructor, move assignment operator)

Constructors

- Must be specified with *public* access.
- Initializes data members in various ways.

```
class class_name{
  public:
    class_name(): member_initializer_list{
      // Constructor Body
    }
};
```

```
class S {
  public:
    int n;
    S() { n = 7;}
};
```

```
#include <iostream>
int main(){
   S s;
   cout << s.n;
}
```

Initialization in Constructor Body

- Data member can be initialized with assignments inside constructor body.

```
class Rectangle {
    int width,height;
 public:
    Rectangle(int x, int y) {
        width = x; height = y;
    }
    int area() { return width * height; }
};
```

```
int main(){
    Rectangle r(3,5);
    cout << r.area() << endl;
}
```

```
15
```

Default Initializer

- Member data could be initialized directly in its declaration with a brace or equals (=).

```
#include <iostream>
#include <string>
class S {
   public:
      int n = 7;
      string s1{“abc”};
      float f = 3.141592;
      string s2 = "DEF";
};
```

```
int main(){
   S s;
   cout << s.n <<
           s.s1 << endl;
   cout << s.f << s.s2;
}
```

```
7abc
3.14159DEF
```

Member Initializer Lists

- It is comma-separated lists after function signature.
  - Member initializer list is processed before the function body of the constructor is executed.

```
class Rectangle {
   int width, height;
 public:
   Rectangle(int x, int y) : width(x), height(y) { }
   int area() { return width * height; }
};
```

```
int main(){
   Rectangle r(3,5);
   cout << r.area() << endl;
}
```

```
15
```

Member Initializer Lists

- If the parameter name and data member name is same, then this-> must be specified as

```cpp
class X {
   public:
      int a, b, i, j;
      X(int i) : b(i), i(i-1), j(this->i){ }
      // i is different from this->i
};
```

```cpp
int main(){
   X x(9);
   cout << x.b << x.i
        << x.j << endl;
}
```

```
988
```

Member Initializer Lists

- When both default initializer and a member initializer list exist for a data member:
  - Member initializer list is executed.
  - default member initializer is ignored.

```
class S {
    int n = 42;
    S() : n(7) {}
};
```

```
#include <iostream>

int main(){
    S s;
    cout << s.n << endl;
}
```

```
7
```

Overloading Constructors

- Constructors can be overloaded to take different parameters.

```
class Rectangle {
   int width, height;
 public:
   Rectangle ();
   Rectangle (int, int);
   int area (void) {
      return (width * height);
    }
};
Rectangle::Rectangle () {
   width = 5; height = 5;
}
Rectangle::Rectangle (int a, int b) {
   width = a; height = b;
}
```

```
int main () {
   Rectangle rect (3,4);
   Rectangle recb;
   cout<<"rect area:"
        <<rect.area()<< endl;
   cout<<"recb area:"
        <<recb.area()<< endl;
   return 0;
}
```

```
rect area:12
recb area:25
```

Delegating Constructor

- A constructor can be used by other constructors in their member initializer list.
  - The list should contain only that constructor.

```
class Foo {
  public:
     char c_in;
     int y_in;
     Foo(char x, int y) {
        c_in = x; y_in = y;
     }
     Foo(int y) : Foo('D', y) {}
};
```

```
int main(){
  Foo f(6);
  cout<<f.c_in
       <<f.y_in<<endl;
}
```

```
D6
```

Default Constructors

- A default constructor is a constructor which can be called with no arguments.
  - Empty parameter list, or
  - Default arguments provided for every parameter

```
class A{
  public:
    int x;
    A(){ x = 1; }
};
class B{
  public:
    int x;
    A(int x = 1): x(x) {}
};
```

```
int main(){
   A a;
   B b;
   cout <<a.x
        <<b.x << endl;
}
```

```
12
```

Implicit Default Constructor

- If no user-declared constructors are provided for a class, the compiler will declare a default constructor with empty function body.

```
#include <iostream>
#include <string>
class S {
  public:
     int n;
     int a[2] = {1, 2};
     string s = "Hello";
};
```

```
int main(){
   S s; // S::S() called
   cout << s.s << endl;
   cout << s.a[0] << ","
        << s.a[1] << endl;
}
```

```
Hello
1,2
```

Destructors

- A destructor is called when the lifetime of an object ends.
  - e.g. program termination, end of scope, etc.
- The destructor is used to free the resources that the object may have acquired during its lifetime.

```
class class_name{
   // Other members
   ~class_name(){
      // Destructor Body
   }
};
```

Destructors

- The destructor may also be called directly with ‘delete’, if the object was created with ‘new’.

```
class A {
 public:
   int i;
   A (int i): i(i) {
      cout<<"c"<<i<<' ';
   }
   ~A() {
      cout<<"d"<<i<<' ';
   }
};
```

```
c0 c1 c2 c3 d2 d3 d1 d0
```

```
A a0(0);
int main()
{
   A a1(1);
   A* p;
   { // nested scope
      A a2(2);
      p = new A(3);
   } // a2 out of scope
   delete p;
// calls the destructor of a3
}
```

Implicit Destructors

- If no user-declared destructor is provided for a class type, the compiler will always declare a destructor with empty function body.

```cpp
#include <iostream>
class S {
public:
    int n = 42;
    S() : n(7) {}
};
```

```cpp
#include <iostream>
int main(){
    S* s = new S();
    cout << s.n << endl;
   delete s; // ~S() called
}
```

```
7
```

Destruction Sequence

- After the body of the destructor is executed, destructors for all non-static members of the class run in reverse order of Constructor.

```
class A {
 public:
   A(){ cout << "C(A)" << endl; }
   ~A(){ cout << "D(A)" << endl; }
};
class B {
 public:
   B(){ cout << "C(B)" << endl; }
   ~B(){ cout << "D(B)" << endl; }
};
```

```
class C{
   A a; B b;
}
int main(){
   C c;
}
```

```
C(A)
C(B)
D(B)
D(A)
```

Objects Defined in Global Scope

- It is constructed at the start of the program, and destructed at the program termination.

```
class A {
 public:
   A(){
      cout << "C(A)" << endl; }
   ~A(){
      cout << "D(A)" << endl;
    }
};
A global_a;
```

```
int main(){
   cout << "Main function
       called"<< endl;
   return 0;
}
```

```
C(A)
Main function called
D(A)
```

Copy Constructor

- It creates an object based on an object of the same class, which has been created previously.

```
// Syntax
class class_name{
 class_name(
   class_name& other){
// Copy Constructor Body
 }
};
```

```
class A{
public:
 int n;
 A(int n = 1) : n(n) { }
 A(const A& a) : n(a.n) { }
};
```

```
int main(){
 A a1(7); A a2(a1);
 cout << a2.n << endl; // 7
}
```

Copy Constructor

- Copy constructor is called when an object is initialized from another object of the same type:
  - Initialization:

`T a = b;` or `T a(b);`, where b is of type T.

  - Function argument passing by value :

`f(a);` , where a is of type T and f is `void f(T t)`.

  - Function return by value :

`return a;` inside a function like `T f()`, where a is of type T.

Implicit Copy Constructor

- If no user-defined copy constructor is provided for a class, the compiler will automatically generate a copy constructor of the form T::T(const T&)
  - The generated constructor performs full member-wise copy of the object's members.

```
#include<iostream>
class A{
 public:
   int m, n;
   A(int n = 1) : n(n) {
      m = 2;
   }
};
```

```
int main(){
   A a1(7);
   A a2(a1);
   cout << a2.m
        << a2.n << endl;
}
```

```
27
```

Copy Assignment Operator

- For an assignment, the copy assignment operator is called instead of copy constructor.
  - Note: instantiation is different from the assignment.

```
class Test {
 public:
   Test() {}
   Test(const Test &t) {
       cout<<"CopyCon"<<endl;
   }
   Test& operator = (const Test &t){
       cout<<"Assign"<<endl;
       return *this;
   }
};
```

```
int main()  {
 Test t1, t2;
 t2 = t1;
 // assignment
 Test t3 = t1;
 // instantiation
 return 0;
}
```

```
Assign
CopyCon
```

Access Specifiers

- Define the accessibility of class members.
- Every member has "access" scope.
    - Default access scope of a member is *private*.

```
class className {
  // default members
public:
  // public members
protected:
  // protected members
private:
  // private members
};
```

```
class S {
   int n;
   // default data member
public:
   void f(); // public function
   S(); // public constructor
private:
   int* ptr;
   // private data member
};
```

Public Member Access

- A public member is accessible everywhere.

```
class Rectangle {
   int width, height;
public:
   Rectangle(int x, int y) {
      width = x; height = y;
   }
   int area() {
      return width*height;}
};
```

```
class Shape{
public:
   Shape(Rectangle r){
      cout << r.area() + 1
           << endl;
    }
};
```

```
int main(){
   Rectangle r(3,5);   Shape s(r);
   cout<<r.area()<<endl;
}
```

```
16
15
```

Protected Member Access

- Protected members are accessible from
  - Other members of the same class.
  - Members of their subclasses.

```
class Base {
protected:
    string s = "Base";
};
class Derived : Base {
public:
   void print(){
      cout << s << endl;
   }
};
```

```
int main(){
   Derived d;
   d.print();
}
```

```
Base
```

Private Member Access

- A private member can only be accessed by the members and ‘friends’ of that class.
  - The concept of friends will be explained later.

```
class Private {
private:
    int n;
public:
    Private() : n(10) {}
    void print(){
        cout << n << endl;
    }
};
```

```
int main(){
    Private p;
    // cout<<p.n<<endl;
    // Compile Error
    p.print();
}
```

```
10
```

Default Member Access

- All default members of class are private accessible only.

```
#include<iostream>
using namespace std;
class Default {
// default
    int n;
public:
    Default() : n(10) {}
    void print(){
        cout<<n<<endl;
    }
};
```

```
int main(){
    Default d;
    // cout<<d.n<<endl;
    // Compile Error
    d.print();
}
```

```
10
```

Struct

- A type defined with the keyword *struct* has public access for its default members.
  - Unlike class, struct is not secure and cannot hide its implementation details from the end user.

```
// Syntax
struct class_name {
  access_specifier :
    member1;
  access_specifier:
    member2;
  ...
};
```

```
struct Courses
{
  char  WebSite[50];
  char  Subject[50];
  int   Price;
};
```

Struct

- In C, struct was used often to group multiple variables since there were no alternatives.
- Use structs as plain-old-data structures.
  - with only a few data members
  - without any class-like features like member functions.

```
struct Point {
   int x,y;
};
```

```
int main(){
   Point p;
   p.x = 3; p.y = 4;
   cout << "(" << p.x
        <<","<< p.y << ")" << endl;
}
```

```
(3,4)
```

Friends

- Keyword *friend* grants a function or another class access to private and protected members.

```
#include <iostream>
class B;
class A {
public:
   void showB(B&);
};
class B {
private:
   int b;
public:
   B() { b = 0; }
   friend void A::showB(B& x);
};
```

```
void A::showB(B& x){
 // it can access private
members of B
   cout << "B::b = " << x.b;
}
```

```
int main() {
   A a;
   B x;
   a.showB(x);
   return 0;
}
```

```
B::b = 0
```

Friends

- A class can be a friend of another class.
- The friend's members can access private and protected members of this class.
- Friendship is not transitive.

```
class A {
   int a = 0;
 public:
   friend class B;
};
class B {
 public:
   void showA(A& x){
       cout << "A::a=" << x.a;
   }
};
```

```
int main() {
   A a;
   B b;
   b.showA(a);
   return 0;
}
```

```
A::a=0
```

Nested Class

- Nested class is a class which is declared in another enclosing class.
  - It has the same access rights as any other members.

```
class Enclosing {
   int x = 987;
public:
   class Nested {
   public:
      void NestedFun(Enclosing *e){
         cout<<e->x;
      }
   } n;
};
```

```
int main() {
  Enclosing e;
  e.n.NestedFun(&e);
  return 0;
}
```

```
987
```

Static Members

- keyword *static* declares members not to be bound to class instances but to the class itself.
- It cannot be initialized inside non-static member functions.

```
class Worker{
public:
  Worker(){
     cout<<"Worker ID : "
         << total << endl;
    total++;
  }
  static int total;
};
```

```
int Worker::total = 0;
int main(){
   Worker w1, w2, w3;
}
```

```
Worker ID : 0
Worker ID : 1
Worker ID : 2
```

Static Member Functions

- There is only one instance of static functions.
  - this pointer cannot be used inside the static function.

```cpp
class Counter {
public:
    static void inc();
    static int count;
};
int Counter::count = 0;
void Counter::inc() {
    count++;
    cout << "Count :"
         << count << endl;
}
```

```cpp
int main(){
    Counter c1,c2,c3;
    c1.inc();
    c2.inc();
    c3.inc();
}
```

```
Cur Count : 1
Cur Count : 2
Cur Count : 3
```

Accessing Static Members

- A static member (m) of a class T can be accessed
  - with the scope operator (::), T::m.
- Can be accessed by an object (o) or a pointer (p)
  - with the member access operators, o.m or p->m .

```
class StaticTest{
public:
   static void f();
   static int n;
};
int StaticTest::n = 7;
void StaticTest::f() {
   cout << n << endl;
   n = 1;
}
```

```
int main(){
 StaticTest::f();
 StaticTest x;  x.f();
 StaticTest* x2 = new StaticTest();
 x2->f(); delete x2;
}
```

```
7
1
1
```

Constant Static Members

- If an int-type static data member is declared const, it can be initialized inside the class body with constant.

```
#include <iostream>
using namespace std;
class Worker{
public:
    Worker();
    static int num_workers;
    const static int MAX_WORKERS = 2;
};
int Worker::num_workers = 0;
```

Constant Static Members

- If an int-type static data member is declared const, it can be initialized inside the class body.

```
Worker::Worker() {
    if(num_workers < MAX_WORKERS){
        cout << "Worker ID : "
             << num_workers << endl;
        num_workers++;
    }
    else {
        cout << "Unable to
             hire new Worker"<<endl;
    }
}
```

```
int main(){
   Worker w1,w2,w3;
}
```

```
Worker ID : 0
Worker ID : 1
Unable to hire new
Worker
```