Basics of C++ Programming
Lecture Notes

BM 531
Numerical Methods and C/C++ Programming

Ahmet Ademoglu, *PhD*
Bogazici University
Institute of Biomedical Engineering

Object Oriented Programming

1. Encapsulation (Data+Methods put together in the class structure)
2. Inheritance (Hierarchy among classes)
3. Polymorphism (The same name or symbol can be used multiple times within different contexts)

The concept which has all these three features is called a **class**. Object Oriented Programming is about developing classes.

Input, Output

To write something on a screen or read from a keyboard, `iostream` **class** is used

```
#include<iostream>
using namespace std; // This is for standard input / output i.e. screen / keyboard
int main(){
int num;
cout << "Hello World " << endl;
cout << "Enter an integer number from the keyboard please : " << endl;
cin >> num;
cout << "The integer  you entered is : " << num << endl;
}
```

A Class Example : Point

enum is a special type that represents a group of constants (unchangeable values) in C/C++.

```
#include <iostream>
using namespace std;
enum Boolean {FALSE, TRUE};
class Point {
public:
int X, Y;
Boolean Visible;
int GetX(){return X;};
Point (int NewX, int NewY); // Constructor Declaration
};
Point::Point (int NewX, int NewY){ // Constructor Definition
X=NewX;
Y=NewY;
Visible=FALSE;
};
int main(){
Point Origin(1,1);
cout << "X Coor=" << Origin.X << "  Y Coor=" << Origin.Y <<"\n";
cout << "X Value by Get function=" << Origin.GetX() << "\n";
return 0;
}
```

Class Constructor

```
#include <iostream>
using namespace std;
enum Boolean {FALSE, TRUE};
class Point {
public:
int X, Y;
Boolean Visible;

int GetX(){return X;};
Point (int NewX, int NewY){ // Constructor Definition & Declaration
X=NewX;
Y=NewY;
Visible=FALSE;
};
Point(int NewX){
X=NewX;
Y=0;
Visible=FALSE;
};
};
int main(){
Point Origin(1,1);
cout << "X Coor=" << Origin.X << "  Y Coor=" << Origin.Y <<"\n";
cout << "X Value by Get function=" << Origin.GetX() << "\n";
Point XPoint(5);
cout << "X Coor=" << XPoint.X << "  Y Coor=" << XPoint.Y <<"\n";
return 0;
}
```

Inheritance

private : can be accessed only by member functions declared within the same class

protected : can be accessed only by member functions within the same class and by member functions of classes that are derived from this class

public : can be accessed from anywhere within the same scope as the class definition

Point Class

```
#include <iostream>
using namespace std;
class Point  {
public:
Point(int = 0, int = 0); // default constructor
void setPoint(int , int ); // set coordinates
int getX() const {return x;} // get X coordinate
int getY() const {return y;} // get Y coordinate
protected :  // accesible by derived class
int x,y; // x and y coordinates of Point
};
Point::Point(int a , int b){
x=a; y=b;
}
void Point::setPoint(int a , int b){
x=a; y=b;
}
```

Circle Class Inherited from Point

```
class Circle : public Point { // Circle inherits from Point
public :
Circle(int r=0, int x=0, int y=0); // default constructor
void setRadius(int); // set radius
int getRadius() const; // return radius
float area() const; // calculate area
protected :
int radius;
};
Circle::Circle (int r, int a, int b): Point(a,b) {// call base class constructor
radius = r;
}
void Circle::setRadius(int r){radius = r;}  //set radius of Circle
int Circle::getRadius() const {return radius;}  //get radius of Circle
float  Circle::area(void) const { return 3.14*radius*radius;}
int main(){
Point mypoint(3,5);
Circle mycircle (1,4,6);
cout << "mypoint x and y coords are " << mypoint.getX() << " and " << mypoint.getY() << '\n';
cout << "mycircle radius, x and y coords are " << mycircle.getRadius() << ", " <<
mycircle.getX() << " and " << mycircle.getY() << '\n';
return 0;
}
```

Inheritance Rules

Access in base class	Access Modifier	Inherited access in base
public	public	public
private	public	not accessable
protected	public	protected
public	private	private
private	private	not accessable
protected	private	private

- Member function **Circle::Circle** needs to access **x** and **y** from base its class **Point**.
- **x** and **y** are protected in **Point** and **Circle** is derived **public** from **Point**.
- **x** and **y** are **protected** within **Circle** and is accessible just like **radius**.
- If **x** and **y** had been defined as **private** in **Point** it would have been inaccessible to member functions of **Circle**.
- However, if you make **x** and **y** **public** in **Point** then **x** and **y** would be accessible by non-member functions as well.
- Since **x** and **y** are protected in **Point** and **Circle** is derived **public** from **Point**, member functions of **Circle** can access to **x** and **y** without exposing it to **public** abuse.

const keyword in functions

```
#include <iostream>
using namespace std;
class Test {
int value;
public:
Test(int v = 0) { value = v; }  // Constructor
// We get compiler error if we add a line like "value = 100;" in this function.
int getValue() const { return value; }
void setValue(int val) { value = val; }// a nonconst function trying to modify value
};
int main()
{
Test t(20);  // Object of the class T
// non-const object invoking const function, no error
cout << t.getValue() << endl;
// const object
const Test t_const(10);
// const object invoking const function, no error
cout << t_const.getValue() << endl;
// const object invoking non-const function, CTE
// t_const.setValue(15);
// non-const object invoking non-const function, no
// error
t.setValue(12);
cout << t.getValue() << endl;
return 0;
}
```

const keyword in functions

The **const** keyword prefixing the name of a variable indicates that the variable is a constant and must not be modified by the program. If a function's argument is a pointer and if that pointer is declared as constant the function cannot modify the contents of the location referenced by that pointer.

```
#include<iostream>
using namespace std;
class Demo {
int val;
public:
Demo(int x = 0) {
val = x;
}
int getValue() const {
return val;
}
};
int main() {
const Demo d(28);
Demo d1(8);
Demo d2;
cout << "The value using object d : " << d.getValue();
cout << "\nThe value using object d1 : " << d1.getValue();
cout << "\nThe value using object d2 : " << d2.getValue();
return 0;
}
```

const member Functions

Use the **const** keyword after the arguments in the declaration of a member function if that member function does not modify any member variable.
This tells the compiler that it can safely apply this member function to a const instance of this class.
If a member function should not alter any data in the class you should declare that member function as **const** function.

```
size_t length(void) const;
```

This informs the compiler that the `length` function should not alter any variable in the class.

Virtual Destructors

```
#include <iostream>
using namespace std;
class Base{
public:
Base() {cout << "Base: constructor" << endl;}
// destructor should be virtual
~Base(){cout << "Base: destructor" << endl;}
};
class Derived : public Base {
public:
Derived() {cout << "Derived: constructor" << endl;}
~Derived() {cout << "Derived: destructor" << endl;}
};
int  main(){
Base* p_base = new Derived;  // use the object...
delete p_base;  // Now delete the object
return 0;
}
```

The output is
Base: constructor
Derived: constructor
Base: destructor

Virtual Destructors

If the base class destructor is defined as:

```
#include <iostream>
using namespace std;
class Base{
public:
Base() {cout << "Base: constructor" << endl;}
// destructor should be virtual
virtual ~Base(){cout << "Base: destructor" << endl;}
};
class Derived : public Base {
public:
Derived() {cout << "Derived: constructor" << endl;}
~Derived() {cout << "Derived: destructor" << endl;}
};
int  main(){
Base* p_base = new Derived;  // use the object...
delete p_base;  // Now delete the object
return 0;
}
```

The output will be
Base: constructor
Derived: constructor
Derived: destructor
Base: destructor

Multiple Inheritance

```
#include<iostream>
using namespace std;
class Person {
 public:
 Person(int x)  { cout << "Person::Person(int ) called" << endl;} // Constructor
};
class Faculty : public Person {
 // data members of Faculty
 public:
 Faculty(int x):Person(x)   {
 cout<<"Faculty::Faculty(int ) called"<< endl;} // Constructor
};
class Student : public Person {
 public:
 Student(int x):Person(x) {
 cout<<"Student::Student(int ) called"<< endl;} // Constructor
};
class TA : public Faculty, public Student  {
 public:
 TA(int x):Student(x), Faculty(x)   {
 cout<<"TA::TA(int ) called"<< endl; } // Constructor
};
int main()  {
 TA ta1(30);
}
```

Note that **Person** is called twice.

Virtual Inheritance

```
#include<iostream>
using namespace std;
class Person {
public:
Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
Person()  { cout << "Person::Person() called" << endl;   } // Default Constructor
};
class Faculty : virtual public Person {
public:
Faculty(int x):Person(x)   {cout<<"Faculty::Faculty(int ) called"<< endl;}
};
class Student : virtual public Person {
public:
Student(int x):Person(x) {cout<<"Student::Student(int ) called"<< endl; }
};
class TA : public Faculty, public Student  {
public:
TA(int x):Student(x), Faculty(x)   {cout<<"TA::TA(int ) called"<< endl; }
};
int main()  {
TA ta1(30);
}
```

When we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to call the parameterized constructor of the 'Person' class?
The constructor has to be called in 'TA' class.

```
#include<iostream>
using namespace std;
class Person {
public:
Person(int x)  { cout << "Person::Person(int ) called" << endl;   } // Parametrized Constructor
Person()     { cout << "Person::Person() called" << endl;   } // Default Constructor
};
class Faculty : virtual public Person {
public:
Faculty(int x):Person(x)   {
cout<<"Faculty::Faculty(int ) called"<< endl;
}
};
class Student : virtual public Person {
public:
Student(int x):Person(x) {
cout<<"Student::Student(int ) called"<< endl;
}
};
class TA : public Faculty, public Student  {
public:
TA(int x):Student(x), Faculty(x), Person(x) { cout<<"TA::TA(int ) called"<< endl;}
};
int main()  {
TA ta1(30);
}
```

Calling virtual Functions in a Base Class Constructor

```
#include<iostream>
using namespace std;
class Base{
public:
Base(){ cout << "Base: constructor. Calling clone()" << endl; clone();}
virtual void clone(){ cout << "Base::clone() called" << endl;}
};
class Derived: public Base{
public:
Derived(){cout << "Derived: constructor"<<endl; }
void clone(){ cout << "Derived::clone() called" << endl;}
};
int main(){
Derived x;
Base *p = &x;
// call clone thru pointer to class instance
cout << "Calling clone through instance pointer";
cout << endl;
p->clone();
return 0;
}
```

On the Output screen

Base: constructor. Calling clone()
Base::clone() called
Derived: constructor
Calling clone thru instance pointer
Derived::clone() called

When you create an instance of a **derived** class the compiler first calls the constructor of the **base** class.
At this point, the **derived** class is partially initialized.
Therefore, when the compiler calls the **clone() virtual** function it cannnot bind it to the **virtual clone**. It instead calls the **base clone()**.
Once the object is created through a **base** class pointer's invoking the **derived clone** function the last line is produced.
If you call a **virtual** function in a **class** constructor the compiler invokes the **base** class version of the function not the version defined for the **derived** class.

Virtual Base Class

```
#include <iostream>  using namespace std;
class device{
public:
device() {cout << "device: constructor" << endl;}
};
class comm_device: public device{
public:
comm_device() {cout << "comm_device: constructor" << endl;}
};
class graphics_device: public device{
public:
graphics_device(){ cout << "graphics_device: constructor" << endl;}
};
class graphics_terminal: public comm_device, public graphics_device{
public:
graphics_terminal() {cout << "graphics_terminal: constructor"<< endl;}
};
int main(){
graphics_terminal gt;
return 0;
}
```

On the Output screen
device: constructor
comm_device: constructor
graphics_device: constructor
graphics_terminal: constructor

Scope resolution operator ::

```
int AllDone;
void AnyFunction(void)
{
int AllDone;
AllDone=1; // refers to local variable
if (::AllDone) // Refers to global variable
DoSomething();
}
```

Dynamic Binding

```
#include <stdio.h>
class shape{public:
virtual double compute_area(void) const{
printf("Not implemented\n");
return 0.; }
virtual void draw(void) const{}; };
class circle_shape : public shape {private:
double X,Y;
double R;
public:
circle_shape(double x, double y, double radius)  {X=x;Y=y;R=radius;
printf("  circle shape implemented with R=%.2lf \n",R);};
virtual double compute_area(void) const {return 0.;};
virtual void draw(void) const {printf("circle shape is drawn \n");} };
class rectangle_shape : public shape{ private:
double X1,Y1;
double X2,Y2;
public:
rectangle_shape(double x1, double y1, double x2, double y2)
{X1=x1;Y1=y1;X2=x2;Y2=y2;printf("  rectangle shape implemented with (X1,Y1)=(%.2lf,%.2lf) \n",X1,Y1);};
virtual double compute_area(void) const {return 0.;};
virtual void draw(void) const {printf("rectangle shape is drawn \n");} };
int main(){
int i;
shape   *shapes[2];
shapes[0]= new circle_shape(100.,100.,50.);
shapes[1]= new rectangle_shape(10.,20.,30.,40);
for (i=0;i<2;i++)  shapes[i]->draw(); }
```

friend functions

```
#include<iostream>
using namespace std;
class complex{
float real, imag;
public:
friend complex add(complex a, complex b);
friend void print(complex a);
complex(){real=imag=0.0;};
complex(float a, float b){real=a;imag=b;}
};
complex add(complex a, complex b){
complex z;
z.real=a.real+b.real;
z.imag=a.imag+b.imag;
return z;
}
void print (complex a){
cout << a.real << "+" << a.imag << "i ";
}
int main(){
complex a,b,c;
a=complex(1.5,2.1);
b=complex(1.1,1.4);
// print and add functions can be accessed from outside the class
cout << "Sum of "; print(a);  cout << "and "; print(b);
c=add(a,b);
cout << " = "; print(c); cout << endl;
return 0;
```

Overloading the operators

```
#include <iostream>
using namespace std;
class complex {
public:
float real,imag;
complex(float a, float b){real=a; imag=b;};
void print(ostream& s) const{s<<real << "+" << imag << "i";};
};
ostream& operator<<( ostream& os, const complex& z){
z.print(os);
return os;
};

int main(){
complex a(1.5,2.1);
cout << "a = "<< a << endl;
return 0;
}
```

Overloading the Input/Output Operators

```
#include <iostream>
#include <cstring>
using namespace std;
class String{
private : char* str;
int len;
public:
String(){
len = 0;
str = new char[len+1];
};
String(const char *s){
len = std::strlen(s);
str = new char[len+1];
std::strcpy(str,s);
};
String(const String&  other){ // copy constructor
len = other.len;
str = new char[len+1];
std::strcpy(str,other.str);
};
~String(){delete[] str;};
```

```
int length() const {return len;};
const char* c_str() const {return str;};
String operator+(const String& other) const { // overloading + operator for String
int new_len = len + other.len;
char* new_str  = new char [new_len+1];
std::strcpy(new_str, str);
std::strcat(new_str, other.str);
String result(new_str);
delete [] new_str;
return  result ;
};
void print (ostream& os) const{
os << str;
}
String& operator=(const String& s){ //overloading Assignment Operator for  String class
if (this != &s ){
len =s.len;
delete [] str;
str= new char[len+1];
strcpy(str,s.str); }
return *this;} ;
};
ostream& operator<<(ostream& os, String& s){
s.print(os);
return os;}
int main(){
String s1="Hello"; String s2=" world.\n";
String s3 = s1+s2;
String s4=s3;
cout << s3; // how can you make it work like cout << s1+s2;?
cout << "s4 is : " << s4;
return 0;}
```

Operators as functions

```
&x  // x.operator&()
x+y //x.operator+(y)
```

Arguments to operator functions
When declared as a friend the operator function requires all arguments explicitly.
This means to declare operator+ as a friend function of class x, you write
`friend x operator+(x&, x&) // assume x is a class`

Operator + for string class

```
String s1("This"), s2("and that"),s3;
s3=s1+s2;

String::String::operator+(const String& s){
size_t len=_length + s._length;
char *t =new char [len+1];
strcpy(t,p_c);
strcat(t,s.p_c);
String r(t);
delete [] t;
return r;
}
String s1="World!";
String s2="Hello,"+s1; // "Hello".operator+(s1) this is an error
```

Solution is to define a friend function which takes two arguments :
`friend String operator+(const String& s1, const String& s2)`
the compiler converts "Hello,"+s1 to the function call:
`operator+(String("Hello"),s1)`

```
#include <iostream>
#include <cstring>
using namespace std;
class String{
private : char* str;
int len;
public:
String(){
len = 0;
str = new char[len+1];
};
String(const char *s){
len = std::strlen(s);
str = new char[len+1];
std::strcpy(str,s);
};
String(const String&  other){ // copy constructor
len = other.len;
str = new char[len+1];
std::strcpy(str,other.str);
};
~String(){delete[] str;};
int length() const {return len;};
const char* c_str() const {return str;};
void print (ostream& os) const{
os << str;
};
```

```
String& operator=(const String& s){ //Assignment Operator for  String class
if (this != &s ){
len =s.len;
delete [] str;
str= new char[len+1];
strcpy(str,s.str);
}
return *this;
}
friend String operator+(const String& s, const String s1 ){
int len = s.length() + s1.length() ;
char *t = new char [len+1];
std::strcpy(t,s.c_str());
std::strcat(t,s1.c_str());
String r(t);
delete [] t;
return r;
}
}; // end String Class

ostream& operator<<(ostream& os, String& s){
s.print(os);
return os;
}
int main(){
String s1="world";
String s2="Hello "+s1;
cout << s2 ;  // However something like cout << "Hello "+s1; will not work!
return 0;
}
```

```
#include <iostream>
#include <cstring>
using namespace std;
class String{
private : char* str;
int len;
public:
String(){
len = 0;
str = new char[len+1];
};
String(const char *s){
len = std::strlen(s);
str = new char[len+1];
std::strcpy(str,s);
};
String(const String&  other){ // copy constructor
len = other.len;
str = new char[len+1];
std::strcpy(str,other.str);
};
~String(){delete[] str;};
int length() const {return len;};
const char* c_str() const {return str;};
void print (ostream& os) const{
os << str;
};
```

```
String& operator=(const String& s){ //Assignment Operator for  String class
if (this != &s ){
len =s.len;
delete [] str;
str= new char[len+1];
strcpy(str,s.str);
}
return *this;
}
friend String operator+(const String& s, const String s1 ){
int len = s.length() + s1.length() ;
char *t = new char [len+1];
std::strcpy(t,s.c_str());
std::strcat(t,s1.c_str());
String r(t);
delete [] t;
return r;
}
friend void  print (const String& s  ){
cout << s.c_str();
}
}; // end String Class
ostream& operator<<(ostream& os, String& s){
s.print(os);
return os; }
int main(){
String s1="world";
String s2="Hello "+s1;
print(s2+"\n") ;
return 0;
}
```

Referencing

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

int main(){
int i=5;
int *p=&i;
int &r=i;
r+=10;  // adds 10 to i because r is another name for i.
cout << r << endl;
cout << *p;
}
```

```
#include <iostream>
using namespace std;
void twice (int &a){
a*=2;}
int main(){
int x=5;
twice(x);
cout << "x=" << x;  // x prints 10
return 0;
}
```

Using Pointer to Class Members

Pointer to Data Member

```
#include <iostream>
 using namespace std;
 class MyClass {
  public:
  int myData;
  MyClass(int data) : myData(data) {}
 };
 int main() {
  MyClass obj(10);
  int MyClass::*ptr = &MyClass::myData; // Define a pointer to a data member
  // Access the data member using the pointer
  int value = obj.*ptr;
  cout << "Value of myData: " << value <<endl;
  return 0;
 }
```

Using Pointer to Class Members

Pointer to Member Function

```
#include <iostream>
using namespace std;
class MathOperations {
 public:
 int add(int a, int b) {
 return a + b;
 }
};
int main() {
 MathOperations math;
 // Define a pointer to a member function
 int (MathOperations::*addPtr)(int, int) = &MathOperations::add;
 // Call the member function using the pointer
 int result = (math.*addPtr)(5, 3);
 cout << "Result of addition: " << result <<endl;
 return 0;
}
```

Using Pointer to Class Members

Pointer to Object

```
#include <iostream>
using namespace std;
class Addition{ int x,y;
 public:
 void setXY(int a, int b){
 x=a;
 y=b; }
 friend int add(Addition ob);
};
int add(Addition ob){
 int Addition::*xPtr=&Addition::x;
 int Addition::*yPtr=&Addition::y;
 Addition *obPtr=&ob;
 int sum=ob.*xPtr+obPtr->*yPtr;
 return sum;  }
int main(){
 Addition obj1;
 // Define a pointer to a member function
 void (Addition::*setPtr)(int,int)=&Addition::setXY;
 // Call the member function using the pointer
 (obj1.*setPtr)(5,10);
 cout<<"Addition: "<<add(obj1)<<endl;
 Addition *obj1Ptr=&obj1;
 (obj1Ptr->*setPtr)(25,35);
 cout<<"Addition: "<<add(obj1);
 return 0;
}
```

Pointers to Member functions

```
#include <iostream>
using namespace std;
class CommandSet{public:
void help(){cout << "Help" << endl;};void nohelp()
{cout << "No Help" << endl;};//...};
void (CommandSet::*f_help)()=&CommandSet::help;
int main(){
CommandSet set1;
(set1.*f_help)();
f_help=&CommandSet::nohelp;(set1.*f_help)();
return 0;}
```

Pointers as references

```
void swap_int(int &a, int &b){  // instead of void swap_int(int *p_a, int *p_b)
int temp;
temp=a;
a=b;
b=temp;
}
int x=2,y=3;
swap_int(x,y);
```

Member Initializer List

```
class Point{
public:
Point (double _x=0.0, double _y=0.)
x=_x;
y=_y;
}
Point (const Point& p) {x=p.x,y=p.y;}
private:
double x,y;
};
class Line{
public:
Line (const Point& b, Point& e) : p1(b), p2(e) {}
private :
Point p1,p2;
};
Line:Line(const Point& b, Point& e){
p1=b;
p2=e;
}
```

File Input/Output

```
#include <fstream.h>
ifstream ins("infile");
ofstream outs("outfile");
if (!ins) { cerr << "cannot open infile \n"; exit (1);}
```

Alternatively,

```
ifstream ins;
ins.open("infile");
.
.
ins.close();
```

To open a file file in binary format `ifstream ins("infile",ios::binary);`
Some member functions of `fstream ins.eof(); ins.get(); ins.put();`