

OOP with C++ Part 1

Immersive Games Technologies

Dr Ji-Jian Chin

University of Plymouth 2023

Topics for this lecture segment A

- Implement classes in C++ and compare the difference between them and conventional structs in C.
- Define access control over an object's members using friend, public, private and/or protected.
- Call an object's member function with the dot and scope operators.
- Define the use of constructors (default) and destructors for a class.
- Differentiate between static, const and mutable effects when applied to members of a class.

Introduction to C++ Class

- In C++, a `class` is a user defined data type
 - Comparable to other pre-defined types e.g `int`, `char`, `bool`
- A `class` is the blueprint / template / mould from which an object is created (*An object is an instance of a class*)
- A programmer *extends* the C++ language by creating his/her own data types as new classes.
- A `class` is similar to `struct` except that all member data is *private* by default and the existence of functions declared as a member of the class (belongs to the instantiated object).

Defining A Class

- Begins with the keyword `class` followed by the class name.
- By convention (but not a rule), class names begin with uppercase letters.
- Have either member functions (or methods), data members (attributes) or both
- A **semicolon ends the definition of the class**.

```
class ClassName {  
    public:  
        typespecifier pub_dataMember;  
        returntype pub_memberFunction( parameter list );  
    private:  
        typespecifier pri_dataMember;  
        returntype pri_memberFunction( parameter list );  
};
```

Public and Private members

- The keywords `public` and `private` are called **access specifiers**.
 - `public`: members declared as public can be accessed from outside of the class.
 - `private`: members declared as private can only be accessed through the member functions of the class

By default, any members are made implicitly `private` if no specifier is specified.

- Data members *normally* made private.
- Member functions are *normally* made public to form the *interface* objects.

```
class aClass
{
    double a, b, c;
    public:
        double x, y, z;
    private:
        double i, j, k;
};
```

private by default if no declaration

public should come first,

keyword *private* needed for subsequent *private* members

```
class Date {
    public:
        int m_nMonth;
        int m_nDay;
        int m_nYear;
        void SetDate(int nMonth, int nDay, int
nYear) {
            m_nMonth = nMonth;
            m_nDay = nDay;
            m_nYear = nYear;
        }
};
```

public data members and functions

- There are two additional access specifiers that can be used to define access control over an object's data members and functions
 - `protected`
 - used when inheriting functions from base class
 - A base class's protected members can be accessed by members and friends of that base class and members and friends of classes *derived* from that base class.
 - `friend`
 - allows functions defined outside of a class's scope the right to access non-public members of the class.
 - can be applied to functions or classes
 - A function prototype declared inside a class is assigned as a 'friend' type with the keyword `friend`.

```
friend void outerFunc(int privateDataVal);
```

Source: <http://www.learncpp.com/cpp-tutorial/813-friend-functions-and-classes/>

```
class Accumulator
{
private:
    int m_nValue;
public:
    Accumulator() { m_nValue = 0; }
    void Add(int nValue) { m_nValue += nValue; }

    // Make the Reset() function a friend of this class
    friend void Reset(Accumulator &cAccumulator);
};

// Reset() is now a friend of the Accumulator class
void Reset(Accumulator &cAccumulator)
{
    // And can access the private data of Accumulator objects
    cAccumulator.m_nValue = 0;
}
```

Objects and Instances

A class definition MUST come **BEFORE** the declaration or instantiation of any class objects

```
class Point
{
    double a, b, c;
public:
    double x, y, z;
private:
    double i, j, k;
};

int main()
{
    Point p;

    p.x = 1.0;
    p.y = 2.0;
    p.z = 3.0;
    return 0;
}
```

class's name (new data type)

C++ keyword class

by default: private data member

class's *public* data members

class's *private* data members

access specifiers

object's name

public data members are accessible with '.' operator

Creating an Object of a Class

- Also known as **creating an instance** (instantiating) of a class.
- Declaring an object of a class is the same process as declaring a variable of an ordinary data type

e.g.: `int var;` declares *var* as a variable of type `int`

`Point p;` declares *p* as an object of type `Point`

- Object *p* is an *instance* of class *Point*.
- It is also possible to declare an array of objects or pointer of class `Point`.
- e.g.: `Point *pPoint;` // *pPoint* is pointer to `Point` type object
 - `Point pArray[30];` // array of `Point` objects

Class Constructors

- A constructor is a class member function (method) that will be invoked automatically by the compiler whenever an instance of the class is created
 - i.e., `ClassName myClass;`
 - The constructor function `ClassName` is called when the object `myClass` is created
 - Constructors are **used to initialize data members and to do any other pre-processing appropriate for an object's creation.**
- A constructor **MUST have the same name of the class**. A class *may* have more than one constructor so long as they have distinct signatures
- A constructor CANNOT have any return type
- `ClassName(int x)` // constructor overloading
 - `ClassName()`
 - `ClassName(int x, int y)`

A constructor may be defined inside or outside class definition

```
class ClassName{
    public:
        //constructor
        ClassName (parameter_list);
    private:
        //.....
};
```

```

class Point {
public:
//constructor
Point(double xarg,double yarg, double zarg);
//public member functions
    double getX( ) const;
    double getY( ) const;
    double getZ( ) const;
private:
    double x, y, z;
};
// class Point constructor
Point :: Point (double x_arg, double y_arg,
double z_arg) {
    x = x_arg;
    y = y_arg;
    z = z_arg;
}
//class Point member function definitions
double Point :: getX( ) const {return x;}
double Point :: getY( ) const{return y;}
double Point :: getZ( ) const{return z;}

```

Same name

The **Point** constructor with three arguments that initialize the three data members of **class Point**.

```

int main( ){
Point p(1.0,2.0,3.0);
    return 0;
}

```

No need to call setX(), setY(), setZ() anymore!

Default class constructors

- Why constructor? It would be ideal that when an object of a class is declared, the values of its data members are assured.
- This is possible if we define a no-argument default constructor and assign values to the class data members internally:
- The default constructor is a **constructor that can be invoked with no arguments**.
- There is **only one default constructor** per class
- All other constructors are known generically as *parameterized* constructors
- By making class data members private we can prescribe the manner in which an object's data is to be manipulated, i.e., by specifying a public interface for the object.
 - This can be done by:
 - assigning **private** access to a class's data members
 - assigning **public** access to member functions that manipulate those private data members
 - allowing indirect access to data members to ensure data member safety
- This can provide data hiding or *encapsulation* and ensure the detail of a class's implementation is being protected.

```
class Point {
public:
    double x, y, z;
    Point();
};
```

Default Constructor

Default Initialization

```
Point :: Point() {
    x = y = z = 1.0; }

int main() {
    Point p;
    cout<<"Hello"<<p.x<<p.y<<p.z<< endl;
    return 0;
}
```

Output: Hello111

```
class Point {
public:
    void SetX (double x_arg) {x = x_arg;}
    void SetY (double y_arg) {y = y_arg;}
    void SetZ (double z_arg) {z = z_arg;}
    double GetX ( ) {return x;}
    double GetY ( ) {return y;}
private:
    double x, y,z;
    void privfunc(void) {cout<<"private";};
};

int main( )
{
    Point p, q;
    p.SetX (1.0); p.SetY (2.0);
    q.SetX (4.0); q.SetY (5.0);
    p.GetX( );
    q.GetX( );
    p.x = 15.0;
    q.privfunc();
    return 0;
}
```

public function
members

private data
members

public member functions Set and Get
are used to access the private data
members x and y

private data and member functions
access here results in compile errors.

Defining Member Functions

- Member functions are **functions that are defined within the body of class definition**, where it behaves as an inline function.

- The syntax for a member function defined outside a class definition is:

```
returntype className::memberFunction (param_list)
```

- The double colon operator '`::`' is known as the *scope resolution operator* - it resolves function calls by allowing the specification of class/library.

```
float Square::calculateArea(float width, float height) {  
    int area; area= width x height;  
    return area;  
}
```

```
class Point{
public:
    void setX (double x_arg) {x = x_arg;}
    void setY (double y_arg) {y = y_arg;}
    double getX ( ) {return x;}
    double getY ( ) {return y;}
private:
    double x, y;
};
```

Member functions
defined inside the class

```
class Point {
public:
    void setX (double x_arg);
    void setY (double y_arg);
    double getX ( );
    double getY ( );
private:
    double x, y;
};

void Point::setX (double x_arg){x = x_arg;}
void Point::setY (double y_arg){y = y_arg;}
double Point :: getX ( ) {return x;}
double Point :: getY ( ) {return y;}
```

A **Point** class with
member functions
defined *outside* of the
class definition, by using
scope resolution
operator

```

class Calc
{
private:
    int m_nValue;
public:
    Calc() { m_nValue = 0; }
    void Add(int nValue);
    void Sub(int nValue);
    void Mult(int nValue);
    int GetValue() {
        return m_nValue;
    }
};

void Calc::Add(int nValue){
    m_nValue += nValue;
}

void Calc::Sub(int nValue){
    m_nValue -= nValue;
}

void Calc::Mult(int nValue){
    m_nValue *= nValue;
}

```

Outside

```

class Calc
{
private:
    int m_nValue;
public:
    Calc() { m_nValue = 0; }

    void Add(int nValue) {
        m_nValue += nValue; }

    void Sub(int nValue) {
        m_nValue -= nValue; }

    void Mult(int nValue) {
        m_nValue *= nValue; }

    int GetValue() {
        return m_nValue; }
};

```

Inside

Class header files

- Similar to declaring functions in your own libraries, classes can also be placed in header files and later *included* in code.
- This promotes the reuse of classes without having to rewrite code.
- Conventionally, a class is placed in a header file with the same name followed by its definition in a similarly named cpp file.
- e.g. student class → student.h + student.cpp
- Any source code wanting to use this class would then just **#include** the header file as its preprocessor directive.

```
#include "Date.h"
Date::Date(int nMth, int nDay, int nYear) {
    SetDate(nMth, nDay, nYear);
}
void Date::SetDate(int nMth,int nDay,int nYear){
    m_nMonth = nMth;
    m_nDay = nDay;
    m_nYear = nYear;
}
```

Date.cpp

Date.h

```
#ifndef DATE_H // header guards
#define DATE_H
class Date {
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;
    Date() { } // private constructor
public:
    Date(int nMth,int nDay,int nYear);
    void SetDate(int nMth, int nDay, int nYear);

    int GetMonth() { return m_nMonth; }
    int GetDay() { return m_nDay; }
    int GetYear() { return m_nYear; }
};
#endif
```


Calling Member Functions

- An object can call a class' *memberFunction()*, using

`objectcls.memberFunction(parameter_list);`

```
class Point {
public:
    void funcA(double x_arg) {x = x_arg;}
private:
    double x, y, z;
};

int main( ) {
    Point p;    // Create the object p of class Point
    p.funcA(1.0); // Call member function funcA() and
                  // passing the argument value 1
    return 0;
}
```

static Data Members and Functions

- `static` data members of a class are `shared by ALL objects / instances` of the class.
- If a change to a data member incurred by one object, the effect will be reflected directly to other objects under the same class.
 - The `static` data members of an uninitialized object are initialized to zero, automatically.
- `static` member functions can only have access to `static` members: cannot access non-static data members and cannot call non-static member functions.

```

class Point {
public:
    static void Member( );
private:
    static double x, y, z; ←
};

double Point :: x=0;
double Point :: y=1; ←
double Point :: z;

void Point :: Member( ) {
    cout << x << y << z << endl; }

```

The **x**, **y**, and **z** data members of *class Point* are defined as **static**

It is important to note that when a *class's* data members are defined as **static** they are not declared.

Global declarations outside the class must be provided for each static data member of a *class*.

Even though two objects are created, both share the same variable and the resulting output of this is '2'

```

class Something
{
public:
    static int s_nValue;
};

int Something::s_nValue = 1;

int main()
{
    Something cFirst;
    cFirst.s_nValue = 2;

    Something cSecond;
    cout << cSecond.s_nValue;

    return 0;
}

```

```
class Something
{
private:
    static int s_nValue;
public:
    static int GetValue() { return s_nValue; }
};

int Something::s_nValue = 1; // initializer

int main()
{
    cout << Something::GetValue() << endl;
}
```

static keyword here removes the ‘attachment’ of the private function to a particular object. Thus, they **can be called with the scope operator** just like a public function!

They still **MUST access static variables** in the object though

Source: <http://www.learncpp.com/cpp-tutorial/812-static-member-functions/>

const Objects and Data members

- A data member declared as **const** MUST be initialized in the constructor and cannot be changed thereafter.
- **const** class data members or **const** object cannot be altered once initialized – this includes directly changing members or calling on member functions.
- **const class objects can only call const member functions** – a function that guarantees it will not change any class variables or call other non-**const** functions.

```
class Something {
public:
    int m_nValue;
    Something() { m_nValue = 0; }
    void ResetValue() { m_nValue = 0; }
    void SetValue(int nValue) { m_nValue = nValue; }
    int GetValue() { return m_nValue; }
};

int main()
{
    const Something cSomething; // calls default constructor

    cSomething.m_nValue = 5; // violates const
    cSomething.ResetValue(); // violates const
    cSomething.SetValue(5); // violates const

    return 0;
}
```

Source: <http://www.learncpp.com/cpp-tutorial/810-const-class-objects-and-member-functions/>

mutable Data Members

- The mutable keyword is used to allow a data member of an `const` object to cast away the `const` property.
- When a data member is declared `mutable`, then the data member of a `const` object is not constant and can therefore be modified – the remainder of the object remains as `const`.

```
class classX {
public:
    int data;
    const int c_data;
    mutable int m_data;
    //mutable const int mc_data;
    //mutable static int mc_data;

classX( ):data(0),c_data(0),m_data(0) {}
};

int main( ) {
    classX x;
    const classX cx;
    x.data = 1;
    x.c_data = 2;
    cx.data = 3;
    cx.c_data = 4;
    cx.m_data = 5;
    return 0;}
```

`const int c_data;` ← *const data member*

`mutable int m_data;` ← *mutable data member*

`//mutable const int mc_data;` ← *Error: mutable const*

`//mutable static int mc_data;` ← *Error: mutable static*

`classX():data(0),c_data(0),m_data(0) {}` ← *Constructor for initializations*

`classX x;` ← *Via initializer list*

`classX x;` ← *non-const object*

`const classX cx;` ← *const object*

`x.data = 1;` ← *non-const object, non-const member*

`x.c_data = 2;` ← *Error: non-const object, const member*

`cx.data = 3;` ← *Error: const object, non-const member*

`cx.c_data = 4;` ← *Error: const object, const member*

`cx.m_data = 5;` ← *mutable member, const object*

const Member Functions

- Allows a member function access to a class's data members, but prevents the member function from altering the value of a data member, i.e.: Read-Only

```
class Point {  
    public:  double X( ) const;  
    private: double x;  
};  
  
inline double Point::X( ) const  
{ x = x + 1;  
  return x; }  
  
int main( ) {  
    Point p;  
    cout << p.X( ) << endl;  
    return 0;  
}
```

const *member function*

Error: cannot alter the *data member*

Class Destructors

- In C++, the complement of the constructor is the destructor.
 - The class destructor has the same name as the class name and constructor name, but preceded by a tilde (~)
 - **Destructor is called when the object goes out of scope.**
 - Destructors perform termination house-keeping i.e. releasing any allocated memory in an object's instance (previously allocated using new)
- Typically, constructor uses the **new** operator when allocating variables, and the destructor uses **delete** operator.
- There is only **one destructor per class**.
 - No destructor overloading allowed.
 - A destructor does not return a type and has no function arguments.

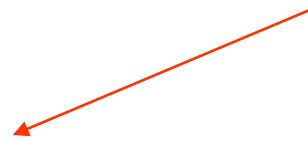
```
class Point {  
    public:  
    Point ( );    // constructor  
    ~Point ( );  // destructor  
};
```



```
class Point {
public:
    int* pArray;    // declare pointer to array
    Point();        // constructor
    ~Point();       // destructor
private:
    float x,y,z;
};
//constructor
Point::Point() :x(0.0), y(0.0), z(0.0) {
    cout << "constructor called" << endl;
    pArray = new int[10]; }
//destructor
Point::~~Point() {
    cout << "destructor called" << endl;
    delete [] pArray; }

int main() {
    Point p;        // create the object, call the constructor
    return 0;
}
```

Initializer list



```
C:\WINDOWS\system32\cmd.exe
constructor called
destructor called
Press any key to continue . . .
```

Break time

Topics for this lecture segment B

By the end of this module, you should be able to:

- Declare class access functions
- Make use of anonymous variables and classes
- Pass objects to methods through value, reference or address
- Control method behaviour with const.
- Use static to declare members and methods of a class
- Use new, new[], delete and delete[] operators to dynamically allocate memory to variables
- Avoid memory leaks by identifying how dynamically allocated memory links can be broken without de-allocation
- Use the hidden this pointer of object classes
- Declare friend methods and classes
- Implement composite and aggregate relationships between classes

Object access functions

- Access functions (getset functions) are simple public functions whose job is simply to return a value (or set a value) of a private data member.
- Typically has two variants : *get* (return a value from private data member) and *set* (set a value of private data member)
- Using access functions **promotes the use of encapsulation** - hiding the details of how something is implemented and instead exposing an interface to the user.
- Encapsulation also **helps prevent accidental changes and misuse** because the member variables can not be accessed directly.

```
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;

public:
    // Getters
    int GetMonth() const { return m_nMonth; }
    int GetDay() const { return m_nDay; }
    int GetYear() const { return m_nYear; }

    // Setters
    void SetMonth(int nMonth) { m_nMonth = nMonth; }
    void SetDay(int nDay) { m_nDay = nDay; }
    void SetYear(int nYear) { m_nYear = nYear; }
};
```

'Get' functions (getters) retrieve data either by returning values or displaying output (e.g. using cout)

'Set' functions set data members, usually one at a time

Anonymous variables

- Anonymous variables are **temporary variables that are not given and identifier/name** (i.e. cannot be referenced by name).
- They have expression scope - once out of an expression, the variable is destroyed - so must be used immediately.

```
int Add(int nX, int nY)
{
    int total=nX+nY;
    return total;
}
```



```
int Add(int nX, int nY)
{
    return nX + nY;
}
```

stored in anonymous
variable then returned

Anonymous classes

- Similar to anonymous variables, class objects can also be anonymous and temporary in nature.
- Also used in place of named class objects that are only used once then destroyed.

```
class Cents {  
private:  
    int m_nCents;  
public:  
    Cents(int nCents) {m_nCents = nCents;}  
    Cents Add(Cents, Cents);  
    int GetCents() { return m_nCents; }  
};  
  
Cents Cents::Add(Cents c1, Cents c2){  
    return Cents(c1.GetCents() + c2.GetCents());  
}  
  
Cents Cents::Add(Cents c1, Cents c2){  
  
    Cents tempcents(c1.GetCents() + c2.GetCents());  
    return tempcents;  
}
```

using anonymous
temporary class to store
value of expression before
return.

using class object to store
value of expression before
return.

Assigning objects

- If both objects are of the same type (that is, both are objects of the same class), then one object can be assigned to another.

```
class Test {  
    int a,b;  
public:  
    void setab(int A, int B){  
        a=A; b=B;  
    }  
};
```

```
int main() {  
    Test obj1, obj2;  
    obj1.setab(5,7);  
    obj2=obj1;  
    return 0  
}
```

C++ built-in assignment operator works on individual class data members

Object passing by value

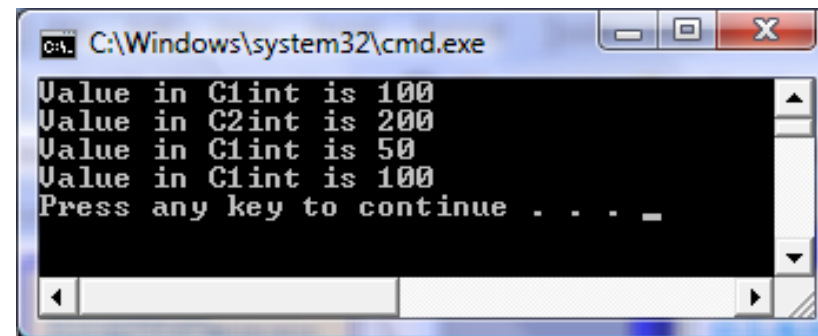
Similar to functions, an entire object can be passed as an argument to methods of a class.

If object is simple/small, the performance hit is negligible. By practice it is **better to use pass by reference** instead.

```
class c1 {  
public:  
    int c1int;  
    c1(){c1int=100;}  
    void c1_function(){  
        cout<<"c1int = "<<c1int;  
    }  
};
```

```
class c2 {  
    int c2int;  
public:  
    c2(){c2int=200;}  
    void c2_function1() {  
        cout<<"c2int = "<<c2int;}  
    void c2_function2(c1 c1obj){  
        c1obj.c1int=50;  
        c1obj.c1_function();  
    }  
};
```

```
int main(){  
    c1 c1object;  
    c2 c2object;  
  
    c1object.c1_function();  
    c2object.c2_function1();  
    c2object.c2_function2(c1object);  
    c1object.c1_function();  
    return 0;  
}
```



```
C:\Windows\system32\cmd.exe  
Value in C1int is 100  
Value in C2int is 200  
Value in C1int is 50  
Value in C1int is 100  
Press any key to continue . . .
```


Object passing by reference

- In passing by reference, only the address of the object is passed to the function.
- Values of object members are directly accessed and can be changed immediately in the function.
- Objects *should* be passed and returned by reference as it is more efficient because there is no need to use temporary variables (anonymous) to store the function arguments.

```
int main() {  
  
    ClassX cx(100);  
  
    ClassY cy;  
  
    cy.doSomething(cx); //pass reference object cx  
  
    cout << cx.get() << endl;  
  
    return 0;  
  
}
```

```
#include <iostream>  
  
using namespace std;  
  
class ClassX {  
public:  
  
    ClassX(int temp) { num=temp; } // Constructor  
  
    void set( int n ) { num = n; }  
  
    int get() { return num; }  
  
private:  
  
    int num;  
  
};  
  
class ClassY {  
public:  
  
    void doSomething( ClassX& ); // pass by reference  
  
};  
  
void ClassY::doSomething(ClassX& c) {  
  
    c.set( -999 );  
  
}
```


Controlling methods with const

- Objects passed by reference to functions suffer from being insecure as they can be modified directly.
- To prevent this, objects passed by reference should be made `const` if the called function should not change the objects' state by setting any of their data members.

Return type Function name (`const` class type);

```
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;
    Date() { } // private default constructor
public:
    Date(int nMonth, int nDay, int nYear){
        SetDate(nMonth, nDay, nYear);
    }
    void SetDate(int nMonth, int nDay, int nYear){
        m_nMonth = nMonth;
        m_nDay = nDay;
        m_nYear = nYear;
    }
    int GetMonth() const { return m_nMonth; }
    int GetDay() const { return m_nDay; }
    int GetYear() const { return m_nYear; }
};
```

'Get' functions should almost always be set as const
to ensure retrieval does not change anything!



Static members of classes

- Any class's static data members and static member functions exist and can be used even if no objects of that class have been instantiated
- A static member function can only access static data members.

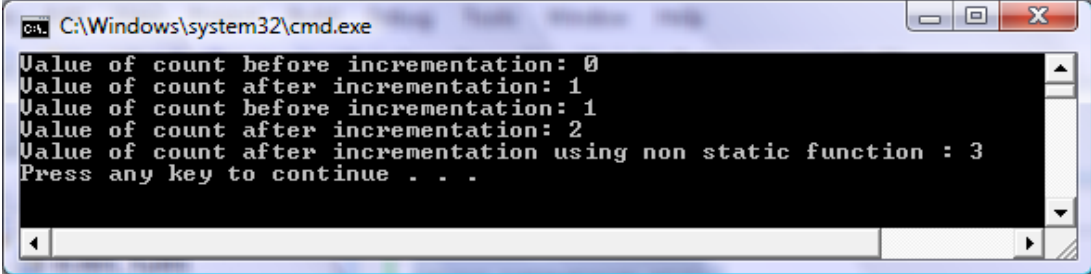
```
#include <iostream>
using namespace std;
```

```
class C {
public:
    static int getCount( ) {return statcount;}
    static void incrstatCount( );
    void incrstatCount2( ){++statcount;}
private:
    static int statcount;
};
```

```
int C::statcount = 0;
void C::incrstatCount( ) {++statcount;}
```

```
int main( ) {
    cout <<"Value of count before incrementation: "<< C::getCount( ) << endl;
    C::incrstatCount( );
    cout <<"Value of count after incrementation: "<< C::getCount( ) << endl;
    C oneC;
    cout <<"Value of count before incrementation: "<< oneC.getCount( ) << endl;
    oneC.incrstatCount( );
    cout <<"Value of count after incrementation: "<< oneC.getCount( ) << endl;
    oneC.incrstatCount2();
    cout <<"Value of count after incrementation using non static function : "<<
        oneC.getCount( ) << endl;

    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Value of count before incrementation: 0
Value of count after incrementation: 1
Value of count before incrementation: 1
Value of count after incrementation: 2
Value of count after incrementation using non static function : 3
Press any key to continue . . .
```

Restricting improper object creation

- Control of proper object creation can be done by declaring the **default constructor as private**.
- This forces new objects to be declared using a parameterized constructor (which can then go through checking before creation).

```
class Employee {  
public:  
    Employee( unsigned ID ) { id = ID; }  
private:  
    Employee( ); // declared private for emphasis  
    unsigned id; // unique id number  
};
```

```
int main( ) {  
    Employee elvis; // ERROR: Employee( ) is private  
    Employee peter(122333); // OK: Employee(unsigned) is public  
    return 0;  
}
```

C++ DMA

- DMA = Dynamic Memory Allocation
- Previously in C, was introduced as `malloc` and `calloc`.
- Instead of declaring a fixed sized variable, C++ uses `new` and `delete` to allocate memory when and where it is needed.

`new` and `new []`

- The `new` operator can be used to define single or multiple **dynamically allocated variables**.
- Single dynamic variable

```
int *ptrInt = new int;  
  
*ptrInt = 100;
```

- Multiple dynamic variable array (calling `new[]` operator)

```
int size=10;  
int *ptrIntArray = new int[size];
```

`delete` and `delete []`

- Variables allocated with `new` can be removed using the `delete` operator.
- Multiple variables declared as a dynamic array MUST use the `delete[]` operator instead.

```
delete ptrInt; // deallocates memory at pointer  
ptrInt=0; // point to nowhere
```

```
delete[] ptrIntArray; // deallocate whole array  
ptrIntArray=0
```

Object passing by address

- Once again, similar to passing variables by address in functions, object can also be passed using pointers.
- Pointers to object typically are used in two contexts in C++
 - a) pointer to object may be passed as arguments to functions or returned by functions
 - b) return or pass objects that have been dynamically allocated using `new` or `new[]` operator.
- The dot operator (`.`) is used exclusively with objects and object references.
- The arrow operator (`->`) is used exclusively with object pointers (also known as dereferencing operator)

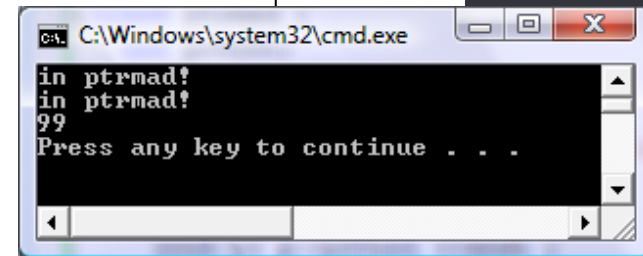
```
class C {
public:
    void m( ) { /* ... */ }
    void f( C* p );
};

void C::f( C* p ) { // receive a pointer to the class C
    p.m( ); // ERROR: p not an object or object reference
    p->m( ); // CORRECT: p is a pointer to a C object
}

int main( ) {
    C c1, c2;
    c1.m( );
    c1.f( &c2 ); // pass the address of object c2
    return 0;
}
```

```
class ptrmad {
    int prvint;
public:
    ptrmad() {prvint=99;}
    void print( ) {cout<<"in ptrmad!\n";}
    void function( ptrmad *p ) {
        p->print();
        cout << ptrmad->prvint ;
    }
};

int main( ) {
    ptrmad ptrobj1,ptrobj2;
    ptrobj1.print();
    ptrobj1.function( &ptrobj2 );
    return 0;
}
```



'this' pointer

- Whenever a method of an object is called, it refers to a hidden **this** pointer to reference its own data members/functions.
- Often the **this** pointer is used to disambiguate similarly named identifiers.
- **this** can also be used to return the entire object from a function after it has completed its task.
- Returning ***this** will return a reference to the object that was implicitly passed to the function by C++.

```
void setID(const string& myID){  
    id = myID;  
}  
  
void setID(const string& myID){  
    this->id = myID;  
}
```

```
class Calc  
{  
private:  
    int m_nValue;  
  
public:  
    Calc() { m_nValue = 0; }  
    Calc& Add(int nValue) { m_nValue += nValue; return *this; }  
    Calc& Sub(int nValue) { m_nValue -= nValue; return *this; }  
    Calc& Mult(int nValue) { m_nValue *= nValue; return *this; }  
    int GetValue() { return m_nValue; }  
};
```

```
class Something  
{  
private:  
    int nData;  
  
public:  
    Something(int nData){  
        this->nData = nData;  
    }  
};
```

friend Functions and Classes

- A friend *function* of a class is defined outside that's class scope, yet has the right to access private members of the class
- Friendship is granted, not taken : for class B to be friend of class A, class A must explicitly declare that class B is its friend
- Friendship is not symmetric : when class B is a friend of class A, it is incorrect to infer that class A is a friend of class B
- Friendship is not transitive : when class C is a friend of class B, and class B is a friend of class A, it is incorrect to infer that class C is a friend of class A


```

#include <iostream>
using namespace std;
class C {
    friend void setX( C &, int );
public:
    C( ) { x = 0; }
    void printX( ) const { cout << x << endl; }
private:
    int x;
};

```

setX is declared to be a friend of class C.
'friend' declarations are neither private nor public

setX can modify private data of C because setX() is declared as a friend function of C

Private data member

```

void setX( C &c, int val ) { c.x = val; }
void cannotSetX( C &c, int val ) { c.x = val; }

```

```

int main( ) {
    C count;
    cout << "count.x after instantiation: ";
    count.printX( );
    cout << "count.x after call to setX( ) friend function: ";
    setX(count, 8 );
    count.printX( );
    cout << "count.x after call to cannotSetX( ) non-friend function: ";
    cannotSetX(count, 18 );
    count.printX( );
    return 0;
}

```

ERROR - cannotSetX cannot modify private data of C because cannotSetX() is not declared as *friend* of C

friend function setX() is allowed to change value of x to 8

ERROR - function cannotSetX() is not allowed to change value of x to 18

To declare a **class's function as friend** in another class

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

```
class F {
public:
    int f( );
```

```
private:
    int adm;
```

```
};
```

```
class C {
```

```
public:
```

```
    int c( );
```

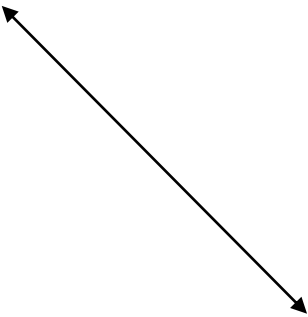
```
    friend int F::f( ); // F's method f is now a friend of class C
```

```
private:
```

```
    int adm;
```

```
};
```

Function f() from class F is declared as a friend function in class C - can access int adm of C



To declare an **entire class as friend** of another class

```
class F {
```

```
    // ...
```

```
};
```

```
class C {
```

```
    friend F; // All methods in F are now friend of class C,
```

```
    // ...
```

```
};
```

Constructor initialization lists

- Constructors have an implicit way of initializing private members with the use of initialization lists.
- Fairly similar to the implicit way of declaring normal variables

e.g. `int newint(5);`

- The initialization list is inserted AFTER the constructor parameters and begins with a colon (:)
- List each variable to be initialized along with the value for that variable separated by a comma.

```
class Point {  
    public:  
    double x, y, z;  
    Point() {  
        x=y=z=0.0;  
    }  
};
```

equivalent

```
class Point {  
    public:  
    double x, y, z;  
    Point() :x(0.0),y(0.0),z(0.0)  
    {}  
};
```

Class composition

- The **process of building complex objects from simpler ones** is called composition.

e.g Car is made up of engine, carburetors, chassis ...

Engine is made up of engine block, pistons, gears,...

- In C++, composition is supported by allowing predefined classes to become member variables of other classes.
- Composition is also supported with the use of initialization lists that allow other objects to be initialized.
- In composition, when the *larger* surrounding object is destroyed, its 'components' are also destroyed (i.e. other objects, data members, functions etc)

- Composition creates “*has-a relationship*” between the objects (e.g. car HAS-A(n) engine).
- The ability of using objects as data members in a class allows the creation of complex structures out of basic data types such as int, char, string etc.

```
#include "CPU.h"  
#include "Motherboard.h"  
#include "RAM.h"
```

} Other object classes written as external header files

```
class PersonalComputer  
{  
private:  
    CPU m_cCPU;  
    Motherboard m_cMotherboard;  
    RAM m_cRAM;  
};
```

} Objects declared as data members in this class

Source: <http://www.learncpp.com/cpp-tutorial/102-composition/>

Initialization of class member variables

- When a class is used as a data member in another class, it's own data members also have to be initialized.
- This can be done by passing arguments through the initializer list in order to initialize the individual objects.

```
PComp::PComp(int nCPUSpeed, char *strMoboMdl, int nRAMSize):  
m_cCPU(nCPUSpeed),m_cMotherboard(strMoboMdl),m_cRAM(nRAMSize)  
{  
}
```

```

#include <iostream>
#include <string>
using namespace std;
Class Date {
public:
    Date( int day_a, int month_a, int year_a );
    void printDate( ) const;
private:
    int day;
    int month;
    int year;
};

class Employee {
public:
    Employee( string, string, int, int, int );
    void printDetail( ) const;
private:
    string firstName;
    string lastName;
    Date birthDate; // composition
};

```

Class object called Date stores a date as 3 integer type variables – day month and year.

Class Employee has constructor with 5 parameters – two for its own private data members, the remainder for the date object

```
Employee::Employee(string fName,string lName,int bDay,int bMonth,int bYear )
: birthDate( bDay, bMonth, bYear )
{
    firstname = fName;
    lastname = lName;
}
```

External constructor definition for
Employee class

```
Date::Date(int inday, int inmonth, int inyear) {
    day = inday;
    month = inmonth;
    year = inyear;
}
```

External constructor definition for
Date class

```
int main( ) {
    Employee e("Bob", "Jones", 24, 7, 1949);
    return 0;
}
```


1-Many Compositional Relationship

- Has-a relationships can typically mirror real world hierarchies.

e.g. CFather has 1 or more CChild objects and 2 House object

```
#include "CChildren.h"
#include "Chouse.h"

class CFather {
public:
    CFather(int numChild, string address1, string address2)
    {
        pChild= new CChildren[numChild];
        house[0].setlocation(address1);
        house[1].setlocation(address2);
    }
private:
    CChildren* pChild; // father has 1 or more child
    Chouse house[2] ; // father has 2 houses
};
```

Recursive composition

- Recursive composition occurs when an object itself becomes its own subobject.
- Objects can be composited recursively using recursive types or references.

```
class List {  
    E value;  
public:  
    List *next, *previous;  
}
```

List class creates a array of List objects (i.e. linked lists)

Why and when to use composition?

- Allows programmers to **assemble complex class structures from simple class members** - makes a class easier to understand as its parts can be broken down and analyzed individually.
- Each sub-object can be self-contained, which makes them reusable.
- The complex class can have the simple subclasses that do most of work, and instead focus on coordinating the data flow between the subclasses.
- There is **no fixed rule as to WHEN to use composition** - but basically if a task can be CLEARLY separated into a standalone task, then it *could* and *should* be a class.

Class aggregation

- Aggregation is a specific variant of composition whereby there is no ownership on an object in a class (in composition, a class declared inside another class is owned by that class)
- Because there is no ownership, when an aggregate object is destroyed, any related sub-objects are NOT destroyed.
- Aggregation takes in other classes as members through the use of **references or pointers that 'point' to objects outside of a class**.
- Because these subclass objects live outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed, but the subclass objects themselves will still exist.

Constructor

Access function

Initialization list

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

class Teacher {
private:
    string m_strName;
public:
    Teacher(string strName)
        : m_strName(strName) {    }
    string GetName() { return m_strName; }
};

class Department {
private:
    Teacher *m_pcTeacher;
public:
    Department(Teacher *pcTeacher=NULL)
        : m_pcTeacher(pcTeacher) {    }
};
```

This department only has one
Teacher object

Constructor

Source: <http://www.learncpp.com/cpp-tutorial/103-aggregation/>

Comparison

- Compositions:
 - Typically use objects as normal member variables
 - Can use pointer values if the composition class automatically handles allocation/de-allocation
 - Responsible for creation/destruction of subclasses
- Aggregations:
 - Typically use pointer variables that point to an object that lives outside the scope of the aggregate class
 - Can use reference values that point to an object that lives outside the scope of the aggregate class
 - Not responsible for creating/destroying subclasses

Thanks so far!

Any Questions?

Reading List



[LearnOpenGL.com](https://learnopengl.com)

Some Exercises:

- <https://coderbyte.com>
- <https://www.codewars.com/>
- <https://leetcode.com>

