Object-Oriented Programming

Classes and Objects

Object-Oriented Programming (OOP)

Content

- Classes and Objects
- Advanced Class Features
- Operator overloading
- Object Relationships
- Abstraction

OOP: Classes and Objects

Content

- Members of the class. Access levels. Encapsulation.
- Class: **interface + implementation**
- Constructors and **destructors**
- **`const`** member functions
- Constructor initializer
- Copy constructor
- Object's lifecycle

OOP: Types of Classes

Types of classes:

- **Polymorphic** Classes – *designed for extension*
  - `Shape`, `exception`,...
- **Value** Classes – *designed for storing values*
  - `int, complex<double>, ...`
- **RAII** (**R**esource **A**cquisition **I**s **I**nitialization) Classes –
- (encapsulate a **resource** into a class → resource lifetime object lifetime)
  - `thread, unique_ptr, ...`

What type of resource?

OOP: Classes and objects

Class = Type (Data + Operations)

- Members of the class
- **Data:**
  - data members (properties, attributes)
- **Operations:**
  - methods (behaviors)
- Each member is associated with an **access level**:
  - `private` **-**
  - `public` **+**
  - `protected` **#**

OOP: Classes and objects

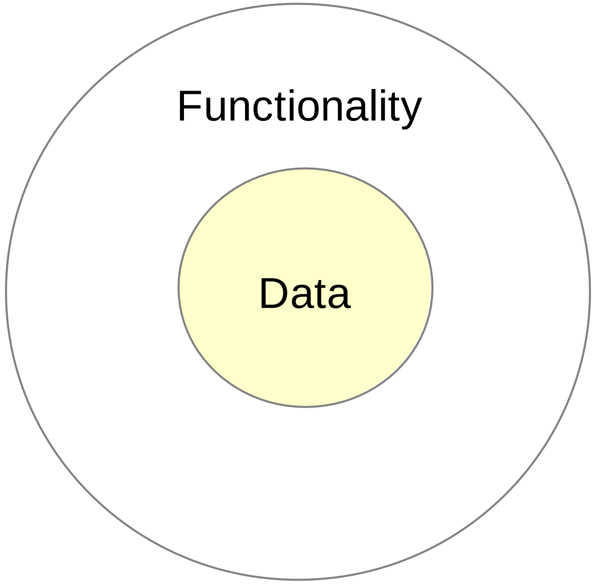
Object = Instance of a class

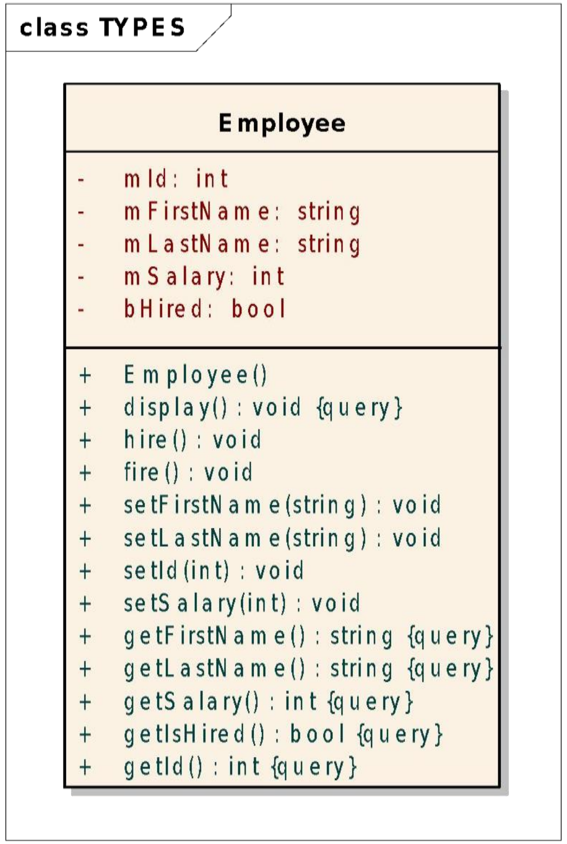
- An employee object: `Employee emp;`
  - **Properties** are the characteristics that describe an object.
    - *What makes this object different?*
      - `id, firstName, lastName, salary, hired`
  - **Behaviors** answer the question:
    - *What can we do to this object?*
      - `hire(), fire(), display(), get` and `set` data members

OOP: Classes and objects

Encapsulation

– an object encapsulates *data* and *functionality*.





OOP: Classes and objects

Class creation

- class **declaration** - *interface*
  - `Employee.h`
- class **definition** – *implementation*
  - `Employee.cpp`

OOP: Classes and objects

Employee.h

```
class Employee{
public:
    Employee();
    void display() const;
    void hire();
    void fire();
    // Getters and setters
    void setFirstName( string inFirstName );
    void setLastName ( string inLastName );
    void setId( int inId );
    void setSalary( int inSalary );
    string getFirstName() const;
    string getLastName() const;
    int getSalary() const;
    bool getIsHired() const;
     int getId() const;
private:
    int mId;
    string mFirstName;
    string mLastName;
    int mSalary;
    bool bHired;
};
```

Methods' declaration

Data members

OOP: Classes and objects

The Constructor and the object's state

- The **state of an object** is defined by its data members.
- The **constructor** is responsible for the **initial state** of the object

```
Employee :: Employee() : mId(-1),
                         mFirstName(""),
                         mLastName(""),
                         mSalary(0),
                         bHired(false){
}
```

Members are initialized through the constructor initializer list

```
Employee :: Employee() {
        mId = -1;
        mFirstName="";
        mLastName="";
        mSalary =0;
        bHired = false;
}
```

Members are assigned

Only constructors can use this initializer-list syntax!!!

OOP: Classes and objects

Constructors

- *responsibility:* data members initialization of a class object
- invoked automatically for each object
- have the *same name* as the class
- have *no return type*
- a class can have *multiple constructors* (function **overloading**)
- may not be declared as `const`
  - constructors can write to const objects

OOP: Classes and objects

Member initialization (C++11)

```
class C{
    string s ("abc");
    double d = 0;
    char * p {nullptr};
    int y[4] {1,2,3,4};
public:
    C(){}
};
```

Compiler

```
class C{
    string s;
    double d;
    char * p;
    int y[5];
public:
    C():s("abc"),
    d(0.0),p(nullptr),
    y{1,2,3,4} {}
};
```

OOP: Classes and objects

Defining a member function

- Employee.cpp
- A **`const`** **member function** cannot change the object's state, can be invoked on const objects

```
void Employee::hire(){
    bHired = true;
}

string Employee::getFirstName() const{
    return mFirstName;
}
```

OOP: Classes and objects

Defining a member function

```
void Employee::display() const {
    cout << "Employee: " << getLastName() << ", "
         << getFirstName() << endl;
    cout << "-------------------------" << endl;
    cout << (bHired ? "Current Employee" :
                      "Former Employee") << endl;
    cout << "Employee ID: " << getId() << endl;
    cout << "Salary: " << getSalary() << endl;
    cout << endl;
}
```

OOP: Classes and objects

TestEmployee.cpp

- Using `const` member functions

```
void foo( const Employee& e){
   e.display();  // OK. display() is a const member function
   e.fire();     // ERROR. fire() is not a const member function
}

int main() {
    Employee emp;
    emp.setFirstName("Robert");
    emp.setLastName("Black");
    emp.setId(1);
    emp.setSalary(1000);
    emp.hire();
    emp.display();
    foo( emp );
    return 0;
}
```

OOP: Classes and objects

Interface: **Employee.h**

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

class Employee{
public:
    Employee();
    //...
protected:
    int mId;
    string mFirstName;
    string mLastName;
    int mSalary;
    bool bHired;
};

#endif
```

Implementation: **Employee.cpp**

```
#include "Employee.h"


Employee::Employee() :
  mId(-1),
  mFirstName(""),
  mLastName(""),
  mSalary(0),
  bHired(false){
}

string Employee::getFirstName() const{
    return mFirstName;
}
/
/ ...
```

OOP: Classes and objects

Object life cycles:

- creation
- assignment
- destruction

OOP: Classes and objects

Object creation:

```
int main() {
    Employee emp;
    emp.display();

    Employee *demp = new Employee();
    demp->display();
    // ..
    delete demp;
    return 0;
}
```

object's lifecycle

- all its *embedded objects* are also created

OOP: Classes and objects

Object creation – constructors:

- *default constructor* (0-argument constructor)

```
Employee :: Employee() : mId(-1), mFirstName(""),
mLastName(""), mSalary(0), bHired(false){
}
```

```
Employee :: Employee() {
}
```

- `Employee employees[ 10 ];`
- `vector<Employee>  emps(10);`

- memory allocation
- constructor call on each allocated object

OOP: Classes and objects

Object creation – constructors:

- *Compiler-generated default constructor*

- if a class *does not specify* any constructors, the *compiler will generate* one that does not take any arguments

```
class Value{
public:
    void setValue( double inValue);
    double getValue() const;
private:
    double value;
};
```

OOP: Classes and objects

Constructors: `default` and `delete` specifiers (C++ 11)

```
class X{
    int i = 4;
    int j {5};
public:
    X(int a) : i{a} {} // i = a, j = 5
    X() = default;       // i = 4, j = 5
};
```

Explicitly forcing the automatic generation of a **default** constructor by the compiler.

OOP: Classes and objects

Constructors: **`default`** and **`delete`** specifiers (C++ 11)

```
class X{
public:
    X( double ){}
};

X x2(3.14); //OK
X x1(10); //OK
```

`int` → `double` conversion

```
class X{
public:
    X( int )= delete;
    X( double );
};
X x1(10); //ERROR
X x2(3.14); //OK
```

OOP: Classes and objects

Best practice: *always provide default values for members!* C++ 11

```
struct
Point{
int x, y;
    Point ( int x = 0, int y = 0 ): x(x), y(y){}
};
class Foo{
    int
    i{};
    double
    d {};
    char c
    {};
    Point p {};
public:
    void print(){
        cout        <<"i:
        "<<i<<endl;   cout
        <<"d:  "<<d<<endl;
        cout        <<"c:
        "<<c<<endl;
        cout <<"p: "<<p.x<<", "<<p.y<<endl;
    }
};
```

```
int main() {
    Foo f;
    f.print(
    );
    return
    0;
}
```

```
OUTPUT:
i: 0
d: 0
c:
p: 0, 0
```

OOP: Classes and objects

Constructor initializer

```
class ConstRef{
public:
    ConstRef( int& );
private:
    int mI;
    const int mCi;
    int& mRi;
};

ConstRef::ConstRef( int& inI ){
    mI = inI;   //OK
    mCi = inI; //ERROR: cannot assign to a const
    mRi = inI; //ERROR: uninitialized reference member
}
```

```
ConstRef::ConstRef( int& inI ): mI( inI ), mCi( inI ), mRi( inI ){}
```

ctor initializer

OOP: Classes and objects

Constructor initializer

- data types that must be initialized in a ctor-initializer
  - `const` data members
  - reference data members
  - object data members having no default constructor
  - superclasses without default constructor

OOP: Classes and objects

A *non-default* Constructor

```
Employee :: Employee( int inId, string inFirstName,
                      string inLastName,
                      int inSalary, int inHired) :
          mId(inId), mFirstName(inFirstName),
          mLastName(inLastName), mSalary(inSalary),
          bHired(inHired){
}
```

OOP: Classes and objects

Delegating Constructor (C++11)

```
class SomeType{
  int number;

public:
  SomeType(int newNumber) : number(newNumber) {}
  SomeType() : SomeType(42) {}
};
```

OOP: Classes and objects

Copy Constructor

```
Employee emp1(1, "Robert", "Black", 4000, true);
```

- called in one of the following cases:
  - `Employee emp2( emp1 ); //copy-constructor called`
  - `Employee emp3 = emp2;  //copy-constructor called`
  - `void foo( Employee emp );//copy-constructor called`
- if you don't define a copy-constructor explicitly, the compiler creates one for you
  - this performs a **bitwise** copy

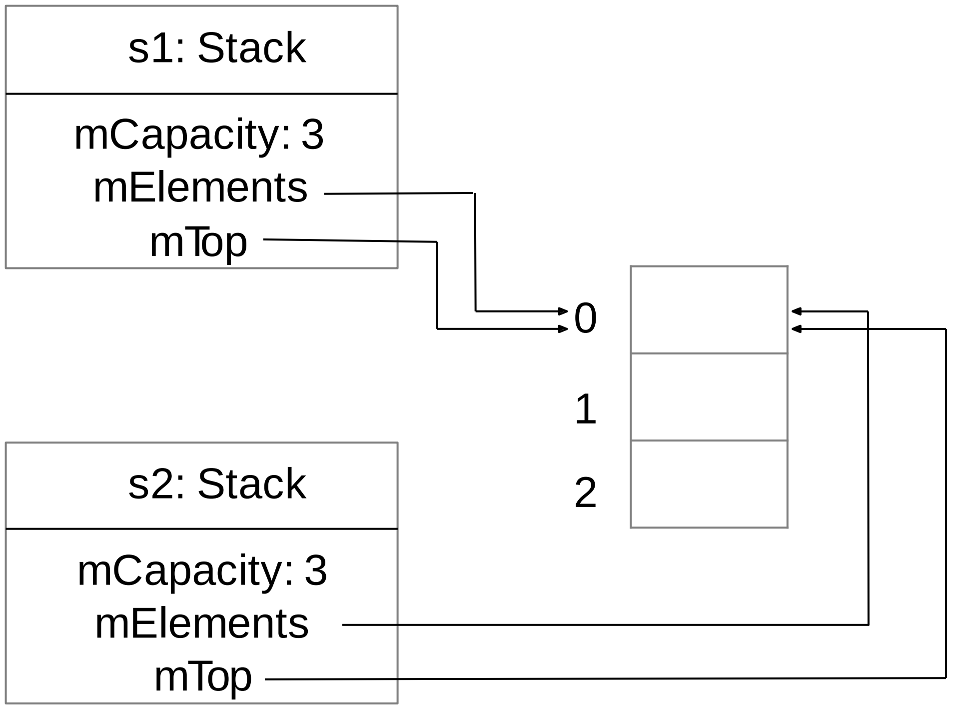
OOP: Classes and objects

```
//Stack.h

#ifndef STACK_H
#define  STACK_H

class Stack{
public:
    Stack( int inCapacity );
    void push( double inDouble );
    double top() const;
    void pop();
    bool isFull() const;
    bool isEmpty()const;

private:
    int mCapacity;
    double * mElements;
    double * mTop;
};
#endif   /* STACK_H */
```

```
//Stack.cpp

#include  "Stack.h"

 Stack::Stack( int inCapacity ){
     mCapacity = inCapacity;
     mElements = new double [ mCapacity ];
     mTop = mElements;
 }

 void Stack::push( double inDouble ){
     if( !isFull()){
         *mTop = inDouble;
         mTop++;
     }
 }
```

OOP: Classes and objects

```
//TestStack.cpp
#include "Stack.h"

int main(){
     Stack s1(3);
    Stack s2 = s1;
    s1.push(1);
    s2.push(2);

    cout<<"s1: "<<s1.top()<<endl;
    cout<<"s2: "<<s2.top()<<endl;
}
```



OOP: Classes and objects

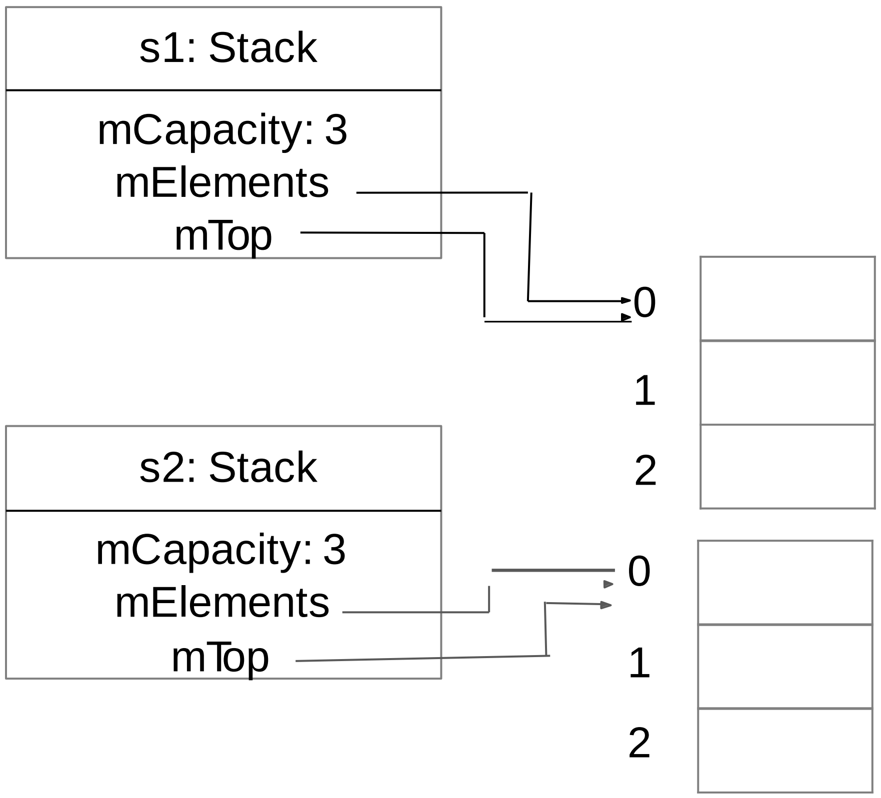
Copy constructor: `T ( const T&)`

```
//Stack.h

#ifndef STACK_H
#define  STACK_H

class Stack{
public:
    //Copy constructor
    Stack( const Stack& );
private:
    int mCapacity;
    double * mElements;
    double * mTop;
};
#endif   /* STACK_H */
```

```
//Stack.cpp

#include "Stack.h"

Stack::Stack( const Stack& s ){
     mCapacity = s.mCapacity;
     mElements = new double[ mCapacity ];
     int nr = s.mTop - s.mElements;
     for( int i=0; i<nr; ++i ){
        mElements[ i ] = s.mElements[ i ];
     }
     mTop = mElements + nr;
 }
```

OOP: Classes and objects

```
//TestStack.cpp
#include "Stack.h"

int main(){
     Stack s1(3);
    Stack s2 = s1;
    s1.push(1);
    s2.push(2);

    cout<<"s1: "<<s1.top()<<endl;
    cout<<"s2: "<<s2.top()<<endl;
}
```



OOP: Classes and objects

Destructor

- when an object is destroyed:
  - the object's destructor is automatically invoked,
  - the memory used by the object is freed.
- each class has one destructor
- usually place to perform cleanup work for the object
- if you don't declare a destructor → the compiler will generate one, which destroys the object's member

OOP: Classes and objects

Destructor

- Syntax: `T :: ~T();`

```
Stack::~Stack(){
    if( mElements != nullptr ){
        delete[] mElements;
        mElements = nullptr;
    }
 }
```

```
{    // block begin
     Stack s(10);            // s: constructor
    Stack* s1 = new Stack(5);// s1: constructor
    s.push(3);
     s1->push(10);
     delete s1;               //s1: destructor
     s.push(16);
}   // block end             //s: destructor
```

OOP: Classes and objects

Default parameters

- if the user specifies the arguments → the defaults are ignored
- if the user omits the arguments → the defaults are used
- the default parameters are specified **only** in the ***method declaration*** (not in the definition)

```
//Stack.h
class Stack{
public:
    Stack( int inCapacity = 5 );
     ..
};
//Stack.cpp
Stack::Stack( int inCapacity ){
     mCapacity = inCapacity;
     mElements = new double [ mCapacity ];
     mTop = mElements;
}
```

```
//TestStack.cpp

Stack   s1(3);   //capacity:   3
Stack   s2;       //capacity:   5
Stack s3( 10 ); //capacity: 10
```

OOP: Classes and objects

The `this` pointer

- every method call passes a pointer to the object for which it is called as *hidden parameter* having the name `this`
- Usage:
  - for disambiguation

```
Stack::Stack( int mCapacity ){
      this → mCapacity = mCapacity;
      //..
}
```

OOP: Classes and objects

Programming task [Prata]

```
class Queue
{
   enum {Q_SIZE = 10};
private:
   // private representation to be developed later
public:
   Queue(int qs = Q_SIZE); // create queue with a qs limit
   ~Queue();
   bool isempty() const;
   bool isfull()  const;
   int queuecount() const;
   bool enqueue(const Item &item); // add item to end
   bool dequeue(Item &item); // remove item from front
};
```

OOP: Classes and objects

Programming task [Prata]

```
class Queue
{
private:
    // class scope definitions

    // Node is a nested structure definition local to this class
    struct Node { Item item; struct Node * next;};
    enum {Q_SIZE = 10};

    // private class members
    Node * front; // pointer to front of Queue
    Node * rear; // pointer to rear of Queue
    int items; // current number of items in Queue
    const int qsize; // maximum number of items in Queue

};
```

Module 3

Object-Oriented Programming

Advanced Class Features

OOP: Advanced class features

Content

- Inline functions
- Stack vs. Heap
- Array of objects vs. array of pointers
- Passing function arguments
- Static members
- Friend functions, friend classes
- Nested classes
- Move semantics (C++11)

OOP: Advanced class features

`Inline` functions

- designed to speed up programs (like macros)
- the compiler replaces the function call with the function code (no function call!)
- advantage: *speed*
- disadvantage: *code bloat*
  - ex. 10 function calls → 10 * function's size

OOP: Advanced class features

How to make a function `inline`?

- use the `inline` keyword either in function declaration or in function definition
- both member and standalone functions can be `inline`
- common practice:
  - place the implementation of the `inline` function into the header file
- only small functions are eligible as `inline`
- the compiler may completely ignore your request

OOP: Advanced class features

`inline` function examples

```
inline double square(double a){
   return a * a;
}

class Value{
   int value;
public:
   inline int getValue()const{ return value; }

    inline void setValue( int value ){
       this->value = value;
   }
};
```

OOP: Advanced class features

- Stack vs. Heap
- Heap – Dynamic allocation

```
void draw(){
    Point * p = new Point();
    p->move(3,3);
    //...
    delete p;
}
```

- Stack – Automatic allocation

```
void draw(){
    Point p;
    p.move(6,6);
    //...
}
```

OOP: Advanced class features

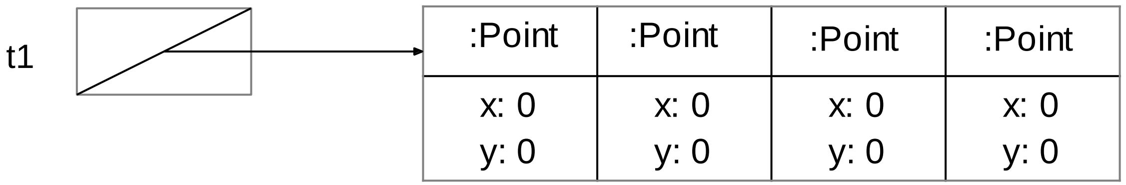
Array of objects

```
class Point{
    int x, y;
public:
    Point( int x=0, int y=0);
    //...
};
```

What is the difference between these two arrays?

Point * t1 = new Point[4];

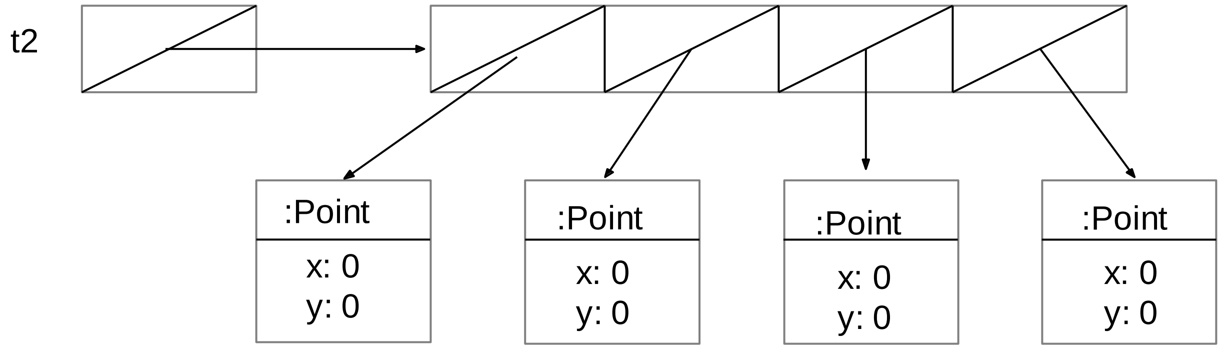
Point t1[4];



OOP: Advanced class features

Array of pointers

```
Point ** t2 = new Point*[ 4 ];
for(int i=0; i<4; ++i ){
     t2[i] = new Point(0,0);
}
for( int i=0; i<4; ++i ){
     cout<<*t2[ i ]<<endl;
}
```



OOP: Advanced class features

Static members:

- `static` methods
- `static` data

- Functions belonging to a *class scope* which don't access object's data can be `static`
- Static methods can't be `const` methods (they do not access object's state)
- They are not called on specific objects ⇒ they have no `this` pointer

OOP: Advanced class features

- Static members

```
//Complex.h

class Complex{
public:
    Complex(int re=0, int im=0);
    static int getNumComplex();
    // ...
private:
   static int num_complex;
   double re, im;
};
```

instance counter

initializing static class member

```
//Complex.cpp

int Complex::num_complex = 0;

int Complex::getNumComplex(){
    return num_complex;
}

Complex::Complex(int re, int im){
    this->re = re;
    this->im = im;
    ++num_complex;
}
```

OOP: Advanced class features

- Static method invocation

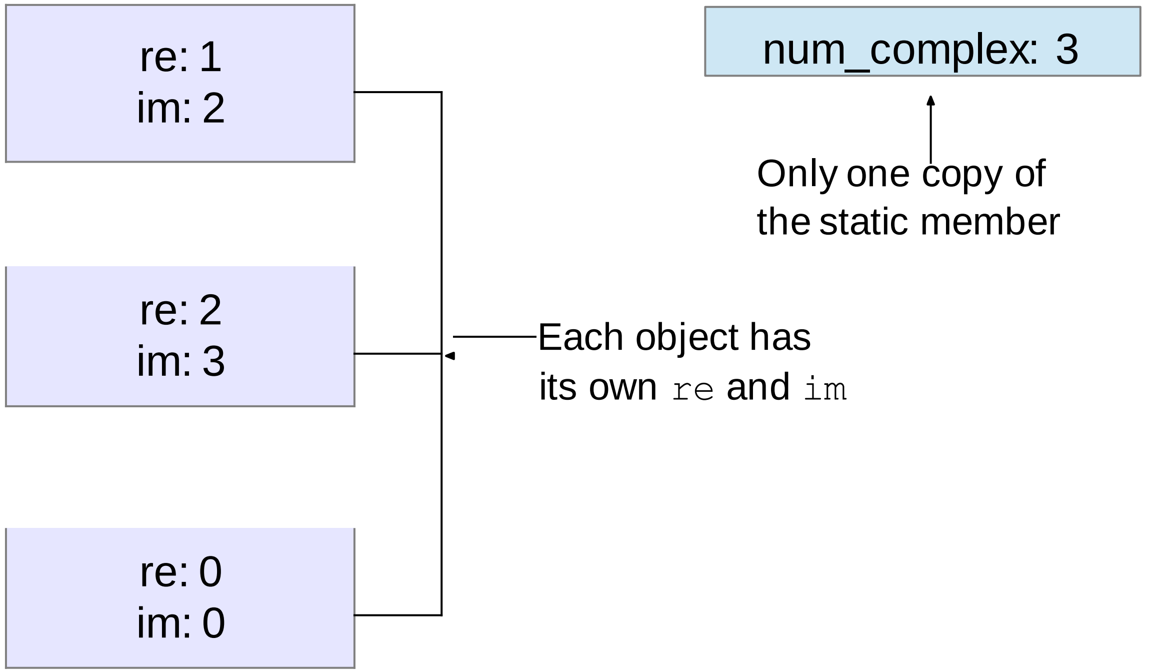
elegant

```
Complex z1(1,2), z2(2,3), z3;
cout<<"Number of complexs:"<<Complex::getNumComplex()<<endl;

 cout<<"Number of complexes: "<<z1.getNumComplex()<<endl;
```

non - elegant

OOP: Advanced class features

Complex z1(1,2), z2(2,3), z3;



OOP: Advanced class features

- Classes vs. Structs
  - default access specifier
    - class: `private`
    - struct: `public`
  - class: data + methods, can be used polymorphically
  - struct: mostly data + convenience methods

OOP: Advanced class features

- Classes vs. structures

```
Class list{
private:
  struct node
  {
    node *next;
    int val;
    node( int val = 0, node * next = nullptr):val(val), next(next){}
  };
  node * mHead;
public:
  list ();
  ~list ();
  void insert (int a);
  void printAll()const;
};
```

OOP: Advanced class features

- Passing function arguments
  - by value
    - the function works on a copy of the variable
  - by reference
    - the function works on the original variable, may modify it
  - by constant reference
    - the function works on the original variable, may not modify (verified by the compiler)

OOP: Advanced class features

- Passing function arguments

passing primitive values

```
void f1(int x)   {x = x + 1;}
void f2(int& x) {x = x + 1;}
void f3(const int& x) {x = x + 1;}// !!!!
void f4(int *x) {*x = *x + 1;}

int main(){
    int y = 5;
    f1(y);
    f2(y);
    f3(y);
    f4(&y);
    return 0;
}
```

OOP: Advanced class features

- Passing function arguments

```
void f1(Point p);
void f2(Point& p);
void f3(const Point& p);
void f4(Point *p);

int main(){
    Point p1(3,3);
    f1(p1);
    f2(p1);
    f3(p1);
    f4(&p1);
    return 0;
}
```

copy constructor will be used on the argument

only `const` methods of the class can be invoked on this argument

OOP: Advanced class features

- `friend` functions, `friend` classes, `friend` member functions
  - friends are allowed to access private members of a class
  - Use it rarely
    - operator overloading

OOP: Advanced class features

- `friend` vs. `static` functions

```
class Test{
private:
    int iValue;
    static int sValue;
public:
    Test( int in ):iValue( in ){}
    void print() const;
    static void print( const Test& what );
    friend void print( const Test& what );
};
```

OOP: Advanced class features

- `friend` vs. `static` functions

```
int Test :: sValue = 0;

void Test::print() const{
    cout<<"Member: "<<iValue<<endl;
}

void Test::print( const Test& what ){
    cout<<"Static: "<<what.iValue<<endl;
}

void print( const Test& what ){
    cout<<"Friend: "<<what.iValue<<endl;
}

int main() {
    Test test( 10 );
    test.print();
    Test::print( test );
    print( test );
}
```

OOP: Advanced class features

- `friend` class vs. `friend` member function

```
class List{
private:
     ListElement * head;
public:
     bool find( int key );
     …
};
```

```
class ListElement{
private:
     int key;
     ListElement * next;
     friend class List;
     ...
};
```

```
Class ListElement{
private:
     int key;
     ListElement * next;
     friend class List::find( int key);
     ...
};
```

OOP: Advanced class features

C++03

- Returning a reference to a `const` object

```
// version 1
vector<int> Max(const vector<int> & v1, const vector<int> & v2){
    if (v1.size() > v2.size())
        return v1;
    else
        return v2;
}

// version 2
const vector<int> & Max(const vector<int> & v1, const vector<int> & v2){
    if (v1.size() > v2.size())
        return v1;
    else
        return v2;
}
```

Copy constructor invocation

More efficient

The reference should be to a non-local object

OOP: Advanced class features

C++11

- Returning a reference to a `const` object

```
vector<int> selectOdd( const vector<int>& v){
   vector<int> odds;
   for( int a: v ){
       if (a % 2 == 1 ){
           odds.push_back( a );
       }
   }
   return odds;
}

//...
vector<int> v(N);
for( int i=0; i<N; ++i){
    v.push_back( rand()% M);
}
vector<int> result = selectOdd( v );
```

EFFICIENT! MOVE constructor invocation

OOP: Advanced class features

- Nested classes
  - the class declared within another class is called a *nested class*
  - usually helper classes are declared as nested

```
// Version 1

class Queue
{
 private:
     // class scope definitions
     // Node is a nested structure definition local to this class
     struct Node {Item item; struct Node * next;};
     ...
};
```

OOP: Advanced class features

- Nested classes [Prata]

Node visibility!!!

```
// Version 2

class Queue
{
     // class scope definitions
     // Node is a nested class definition  local to this class
     class Node
     {
     public:
          Item item;
          Node * next;
          Node(const Item & i) : item(i), next(0) { }
     };
     //...
};
```

OOP: Advanced class features

- Nested classes
  - a nested class **B** declared in a **private** section of a class **A**:
    - **B** is local to class **A** (only class A can use it)
  - a nested class **B** declared in a **protected** section of a class **A**:
    - **B** can be used both in **A** and in the derived classes of **A**
  - a nested class **B** declared in a **public** section of a class **A**:
    - **B** is available to the outside world (Usage: `A::B b;`)

OOP: Advanced class features

- Features of a *well-behaved* C++ class
  - implicit constructor
    - `T :: T(){ … }`
  - destructor
    - `T :: ~T(){ … }`
  - copy constructor
    - `T :: T( const T& ){ … }`
  - assignment operator (*see next module*)
    - `T& T :: operator=( const T& ){ … }`

OOP: Advanced class features

- Constructor delegation (C++11)

```
// C++03
class A
{
    void init() { std::cout << "init()"; }
    void doSomethingElse() { std::cout << "doSomethingElse()\n"; }
public:
    A() { init(); }
    A(int a) { init(); doSomethingElse(); }
};
```

```
// C++11
class A
{
    void doSomethingElse() { std::cout << "doSomethingElse()\n"; }
public:
    A() { ... }
    A(int a) : A() { doSomethingElse(); }
};
```

OOP: Advanced class features

- Lvalues:
  - Refer to objects accessible at more than one point in a source code
    - Named objects
    - Objects accessible via pointers/references
  - Lvalues may not be moved from
- Rvalues:
  - Refer to objects accessible at exactly one point in source code
    - Temporary objects (e.g. by value function return)
  - Rvalues may be moved from

OOP: Advanced class features

- Lvalue

```
int  x;
x = 10;
```

Rvalue

OOP: Advanced class features

- Move Semantics (C++11)

```
class string{
    char* data;
public:
   string( const char* );
   string( const string& );
   ~string();
};
```

```
string :: string(const char* p){
    size_t size = strlen(p) + 1;
    data = new char[size];
    memcpy(data, p, size);
}
string :: string(const string& that){
    size_t size = strlen(that.data) + 1;
    data = new char[size];
    memcpy(data, that.data, size);
}

string :: ~string(){
    delete[] data;
}
```

OOP: Advanced class features

- Move Semantics (C++11): lvalue, rvalue

```
string a(x);                                // Line 1
string b(x + y);                            // Line 2
string c(function_returning_a_string());    // Line 3
```

- **lvalue:** real object having an address
  - **Line 1:** `x`
- **rvalue:** temporary object – no name
  - **Line 2:** `x + y`
  - **Line 3:** `function_returning_a_string()`

OOP: Advanced class features

- Move Semantics (C++11): rvalue reference, move constructor

```
//string&& is an rvalue reference to a string
string :: string(string&& that){
    data = that.data;
    that.data = nullptr;
}
```

- **Move constructor**
  - **Shallow copy** of the argument
  - **Ownership transfer** to the new object

OOP: Advanced class features

- **Move** constructor – Stack class

```
Stack::Stack(Stack&& rhs){
    //move rhs to this
    this->mCapacity = rhs.mCapacity;
    this->mTop = rhs.mTop;
    this->mElements = rhs.mElements;

    //leave rhs in valid state
    rhs.mElements = nullptr;
    rhs.mCapacity = 0;
    rhs.mTop = 0;
}
```

OOP: Advanced class features

- Copy constructor vs. move constructor
  - Copy constructor: **deep copy**
  - Move constructor: **shallow copy + ownership transfer**

```
// constructor
string s="apple";
// copy constructor: s is an lvalue
string s1 = s;
// move constructor: right side is an rvalue
string s2 = s + s1;
```

OOP: Advanced class features

- Passing large objects

```
// C++98
// avoid expense copying

void makeBigVector(vector<int>& out){
 ...
}
vector<int> v;
makeBigVector( v );
```

```
// C++11
// move semantics

vector<int> makeBigVector(){
 …
}
auto v = makeBigVector();
```

- All STL classes have been extended to support **move semantics**
- The content of the temporary created vector is moved in v (not copied)

http://geant4.web.cern.ch/geant4/collaboration/c++11_guidelines.pdf

OOP: Advanced class features

```
class A{
    int value {10};
    static A instance;
public:
    static A& getInstance(){ return instance;}
    static A  getInstanceCopy(){ return instance;}
    int getValue() const { return value;}
    void setValue( int value ){ this->value = value;}
};
```

```
A A::instance;
int main(){
    A& v1 = A::getInstance();
    cout<<"v1: "<<v1.getValue()<<endl;
    v1.setValue(20);
    cout<<"v1: "<<v1.getValue()<<endl;
    A v2 = A::getInstanceCopy();
    cout<<"v2: "<<v2.getValue()<<endl;
    return 0;
}
```

Reference to a static variable → **lvalue**

A temporary copy of instance → **rvalue**

Output?

Module 4

Object-Oriented Programming

Operator overloading

OOP: Operator overloading

Content

- Objectives
- Types of operators
- Operators
  - Arithmetic operators
  - Increment/decrement
  - Inserter/extractor operators
  - Assignment operator (copy and move)
  - Index operator
  - Relational and equality operators
  - Conversion operators

OOP: Operator overloading

Objective

- To make the class usage easier, more intuitive
  - the ability to read an object using the `extractor` operator (>>)
    - `Employee e1; cin >> e;`
  - the ability to write an object using the `inserter` operator (<<)
    - `Employee e2; cout<<e<<endl;`
  - the ability to compare objects of a given class
    - `cout<< ((e1 < e2) ? "less" : "greater");`

Operator overloading: a service to the clients of the class

OOP: Operator overloading

Limitations

- You cannot add new operator symbols. Only the existing operators can be redefined.
- Some operators cannot be overloaded:
  - `.` (member access in an object)
  - `::`(scope resolution operator)
  - **`sizeof`**
  - **`?:`**
- You cannot change the **arity** (the number of arguments) of the operator
- You cannot change the **precedence** or **associativity** of the operator

OOP: Operator overloading

How to implement?

- write a function with the name `operator<symbol>`
- alternatives:
  - method of your class
  - global function (usually a friend of the class)

http://en.cppreference.com/w/cpp/language/operators

OOP: Operator overloading

- There are 3 types of operators:
  - operators that must be methods (**member functions**)
    - they don't make sense outside of a class:
      - `operator=, operator(),operator[], operator->`
  - operators that must be **global functions**
    - the left-hand side of the operator is a variable of different type than your class: `operator<<, operator>>`
      - `cout << emp;`
        - `cout: ostream`
        - `emp: Employee`
  - operators that can be **either** methods or global functions
    - **Gregoire:** “Make every operator a method unless you must make it a global function.”

OOP: Operator overloading

- Choosing argument types:
  - value vs. reference
    - Prefer passing-by-reference instead of passing-by-value.
  - `const` vs. non const
    - Prefer const unless you modify it.
- Choosing return types
  - you can specify any return type, however
    - follow the built-in types rule:
      - comparison always return `bool`
      - **arithmetic operators return an object representing the result of the arithmetic**

OOP: Operator overloading

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex{
public:
    Complex(double, double );
    void setRe( double );
    void setIm( double im);
    double getRe() const;
    double getIm() const;
    void print() const;
private:
    double re, im;
};
#endif
```

OOP: Operator overloading

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

Complex::Complex(double re, double im):re(

re),im(im) {} void Complex::setRe( double re){this-

>re = re;}

void Complex::setIm( double im){ this->im =

im;}  double Complex::getRe() const{ return

this->re;}   double Complex::getIm() const{

return this->im;}

void Complex::print()const{    cout<<re<<"+"<<im<<"i";}
```

OOP: Operator overloading

- Arithmetic operators (**member** or standalone func.)
  - unary minus
  - binary minus

```
Complex Complex::operator-() const{
    Complex temp(-this->re, -this->im);
    return temp;
}

Complex Complex::operator-( const Complex& z) const{
    Complex temp(this->re - z.re, this->im- z.im);
    return temp;
}
```

OOP: Operator overloading

- Arithmetic operators (member or **standalone** func.)
  - unary minus
  - binary minus

```
Complex operator-( const Complex& z ){
    Complex temp(-z.getRe(), -z.getIm());
    return temp;
}

Complex operator-( const Complex& z1, const Complex& z2 ){
    Complex temp(z1.getRe()-z2.getRe(), z1.getIm()-z2.getIm());
    return temp;
}
```

OOP: Operator overloading

– Increment/Decrement operators

- postincrement:

  ```
  - int i = 10; int j = i++; // j → 10
  ```

- preincrement:

  ```
  - int i = 10; int j = ++i; // j → 11
  ```

- The C++ standard specifies that the prefix increment and decrement return an **lvalue** (left value).

OOP: Operator overloading

- Increment/Decrement operators (member func.)

```
Complex& Complex::operator++(){             //prefix
      (this->re)++;
      (this->im)++;
      return *this;
}

Complex   Complex::operator++( int ){ //postfix
      Complex
      temp(*this);
      (this->re)++;
      (this-
      >im)++;
      return
      temp;
}
```

Which one is more efficient? Why?

OOP: Operator overloading

- Inserter/Extractor operators (standalone func.)

```
//complex.h

class Complex {
public:
    friend ostream& operator<<(
                    ostream& os, const Complex& c);
    friend istream& operator>>(
                    istream& is, Complex& c);
    //...
};
```

OOP: Operator overloading

- Inserter/Extractor operators (standalone func.)

```
//complex.cpp

ostream& operator<<( ostream& os, const Complex& c){
     os<<c.re<<"+"<<c.im<<"i";
     return os;
}

istream& operator>>( istream& is, Complex& c){
     is>>c.re>>c.im;
     return is;
}
```

OOP: Operator overloading

- Inserter/Extractor operators
- **Syntax:**

```
ostream& operator<<( ostream& os, const T& out)
istream& operator>>( istream& is, T& in)
```

- Remarks:
  - Streams are always *passed by reference*
  - **Q:** Why should inserter operator return an **ostream&**?
  - **Q:** Why should extractor operator return an **istream&**?

OOP: Operator overloading

- Inserter/Extractor operators
- Usage:

```
Complex z1, z2;
cout<<"Read 2 complex number:";
//Extractor
cin>>z1>>z2;
//Inserter
cout<<"z1: "<<z1<<endl;
cout<<"z2: "<<z2<<endl;

cout<<"z1++: "<<(z1++)<<endl;
cout<<"++z2: "<<(++z2)<<endl;
```

OOP: Operator overloading

- **Assignment operator (=)**
  - **Q:** When should be overloaded?
  - **A:** When bitwise copy is not satisfactory (e.g. if you have dynamically allocated memory ⇒
    - when we should implement the copy constructor and the destructor too).
    - Ex. our Stack class

OOP: Operator overloading

- Assignment operator (member func.)
  - Copy assignment
  - Move assignment (since C++11)

OOP: Operator overloading

- **Copy** assignment operator (member func.)
  - **Syntax:** `X& operator=( const X& rhs);`
  - **Q:** Is the return type necessary?
    - Analyze the following example code

```
Complex z1(1,2), z2(2,3), z3(1,1);
z3 = z1;
z2 = z1 = z3;
```

OOP: Operator overloading

– **Copy** assignment operator example

```
Stack& Stack::operator=(const Stack& rhs) {
 if (this != &rhs) {
   //delete lhs - left hand side
   delete [] this->mElements;
   this->mCapacity = 0;
    this_>melements = nullptr; // in case next line throws
   //copy rhs - right hand side
   this->mCapacity = rhs.mCapacity;
   this->mElements = new double[ mCapacity ];
   int nr = rhs.mTop - rhs.mElements;
   std::copy(rhs.mElements,rhs.mElements+nr,this->mElements);
   mTop = mElements + nr;
 }
 return *this;
}
```

OOP: Operator overloading

- Copy assignment operator vs Copy constructor

```
Complex z1(1,2), z2(3,4); //Constructor
Complex z3 = z1; //Copy constructor
Complex z4(z2);   //Copy constructor
z1 = z2;          //Copy assignment operator
```

OOP: Operator overloading

- **Move** assignment operator (member func.)
  - **Syntax:** `X& operator=( X&& rhs);`
  - When it is called?

```
Complex z1(1,2), z2(3,4); //Constructor
Complex z4(z2); //Copy constructor
z1 = z2;      //Copy assignment operator
Complex z3 = z1 + z2; //Move constructor
z3 = z1 + z1;//Move assignment
```

OOP: Operator overloading

- **Move** assignment operator example

```
Stack& Stack::operator=(Stack&& rhs){
    //delete lhs - left hand side
    delete [] this->mElements;
     //move rhs to this
     this->mCapacity = rhs.mCapacity;
    this->mTop = rhs.mTop;
    this->mElements = rhs.mElements;
    //leave rhs in valid state
    rhs.mElements = nullptr;
    rhs.mCapacity = 0;
     rhs.mTop = 0;
    //return permits s1 = s2 = create_stack(4);
    return *this;
}
```

OOP: Advanced class features

- Features of a *well-behaved* C++ class (**2011**)
  - implicit constructor `T :: T();`
  - destructor `T :: ~T();`
  - copy constructor `T :: T( const T& );`
  - **move** constructor `T :: T( T&& );`
  - copy assignment operator
    - `T& T :: operator=( const T& );`
  - **move** assignment operator
    - `T& T :: operator=( T&& rhs );`

OOP: Operator overloading

- Subscript operator: needed for arrays (member func.)
- Suppose you want your own dynamically allocated C-style array ⇒ implement your own `CArray`

```
#ifndef CARRAY_H
#define   CARRAY_H
class CArray{
public:
    CArray( int size = 10 );
    ~CArray();
    CArray( const CArray&) = delete;
    CArray& operator=( const Carray&) = delete;
    double& operator[]( int index );
    double  operator[]( int index ) const;
private:
    double * mElems;
    int mSize;
};
#endif    /* ARRAY_H */`
```

Provides read-only access

"If the value type is known to be a built-in type, the const variant should return by value."
http://en.cppreference.com/w/cpp/language/operators.

OOP: Operator overloading

- Implementation

```
CArray::CArray( int size ){ if( size
    < 0 ){
        this->size = 10;
    }
    this->mSize = size;
    this->mElems = new double[ mSize ];
}

CArray::~CArray(){
    if( mElems != nullptr ){
        delete[] mElems;  mElems =
        nullptr;
    }
}

double& CArray::operator[]( int index ){ if(
    index <0 || index >= mSize ){
        throw out_of_range("");
    }
    return mElems[ index ];
}
```

```
double CArray::operator[](
                      int index ) const{
    if( index <0 || index >= mSize ){
        throw out_of_range("");
    }
    return mElems[ index ];
}
```

#include<stdexcept>

OOP: Operator overloading

- `const` vs non-const `[]` `operator`

```
void printArray(const CArray& arr, size_t size) {
    for (size_t i = 0; i < size; i++) {
        cout << arr[i] << "" ;
        // Calls the const operator[] because arr is
        // a const object.
    }
    cout << endl;
}
```

```
CArray myArray;
for (size_t i = 0; i < 10; i++) {
    myArray[i] = 100;
    // Calls the non-const operator[] because
    // myArray is a non-const object.
}
printArray(myArray, 10);
```

OOP: Operator overloading

- Relational and equality operators
  - used for **search** and **sort**
  - the container must be able to compare the stored objects

```
bool operator ==( const Point& p1, const Point& p2){
    return p1.getX() == p2.getX() && p1.getY() == p2.getY();
}


bool operator <( const Point& p1, const Point& p2){
   return p1.distance(Point(0,0)) < p2.distance(Point(0,0));
}
```

```
set<Point> p;
```

```
vector<Point> v; //...
sort(v.begin(), v.end());
```

OOP: Operator overloading

- The function call operator `()`
- Instances of classes overloading this operator behave as functions too (they are function objects = function + object)

```
#ifndef ADDVALUE_H
#define   ADDVALUE_H
class AddValue{
    int value;
public:
    AddValue( int inValue = 1);
    void operator()( int& what );
};
#endif    /* ADDVALUE_H */
```

```
#include "AddValue.h"

AddValue::AddValue( int inValue ){
    this->value = inValue;
}

void AddValue::operator()( int& what ){
    what += this->value;
}
```

OOP: Operator overloading

- The function call operator

```
AddValue func(2);
int array[]={1, 2, 3};
for( int& x : array ){
   func(x);
}
for( int x: array ){
   cout <<x<<endl;
}
```

OOP: Operator overloading

- Function call operator
  - used frequently for defining sorting criterion

```
struct EmployeeCompare{
    bool operator()( const Employee& e1, const Employee&
        e2){ if( e1.getLastName() == e2.getLastName())
            return e1.getFirstName() < e2.getFirstName();
        else
            return e1.getLastName() < e2.getLastName();

    }
};
```

OOP: Operator overloading

- Function call operator
  - sorted container

```
set<Employee, EmployeeCompare> s;

Employee e1; e1.setFirstName("Barbara");
e1.setLastName("Liskov");
Employee e2; e2.setFirstName("John");
e2.setLastName("Steinbeck");
Employee e3; e3.setFirstName("Andrew");
e3.setLastName("Foyle");
s.insert( e1 ); s.insert( e2 ); s.insert( e3 );

for( auto& emp : s){
   emp.display();
}
```

OOP: Operator overloading

- Sorting elements of a given *type*:
  - **A.** override operators: **<**, **==**
  - **B.** define a function object containing the comparison
- **Which one to use?**
  - **Q:** How many sorted criteria can be defined using method **A**?
  - **Q:** How many sorted criteria can be defined using method **B**?

OOP: Operator overloading

- Writing conversion operators

```
class Complex{
public:
   operator string() const;
   //
};

Complex::operator string() const{
    stringstream ss;
    ss<<this->re<<"+"<<this->im<<"i";
    return ss.str();
}
```

```
//usage
Complex z(1, 2);
string a = z;
cout<<a<<endl;
```

OOP: Operator overloading

- After templates
  - Overloading operator *
  - Overloading operator →

OOP: Review

- Find all possible errors or shortcommings!

```
class Array {
public:
  Array (int n) : rep_(new int [n]) { }
  Array (Array& rhs) : rep_(rhs.rep_) { }
  ~Array () { delete rep_; }
  Array& operator = (Array rhs) { rep_= rhs.rep_;   }
  int& operator [] (int n) { return &rep_[n]; }
private:
  int * rep_;
}; // Array
```

Source: http://www.cs.helsinki.fi/u/vihavain/k13/gea/exer/exer_2.html

Solution required!

- It is given the following program!

```
#include <iostream>

int main(){
    std::cout<<"Hello\n";
    return 0;
}
```

Modify the program *without modifying the main function* so that the output of the program would be:

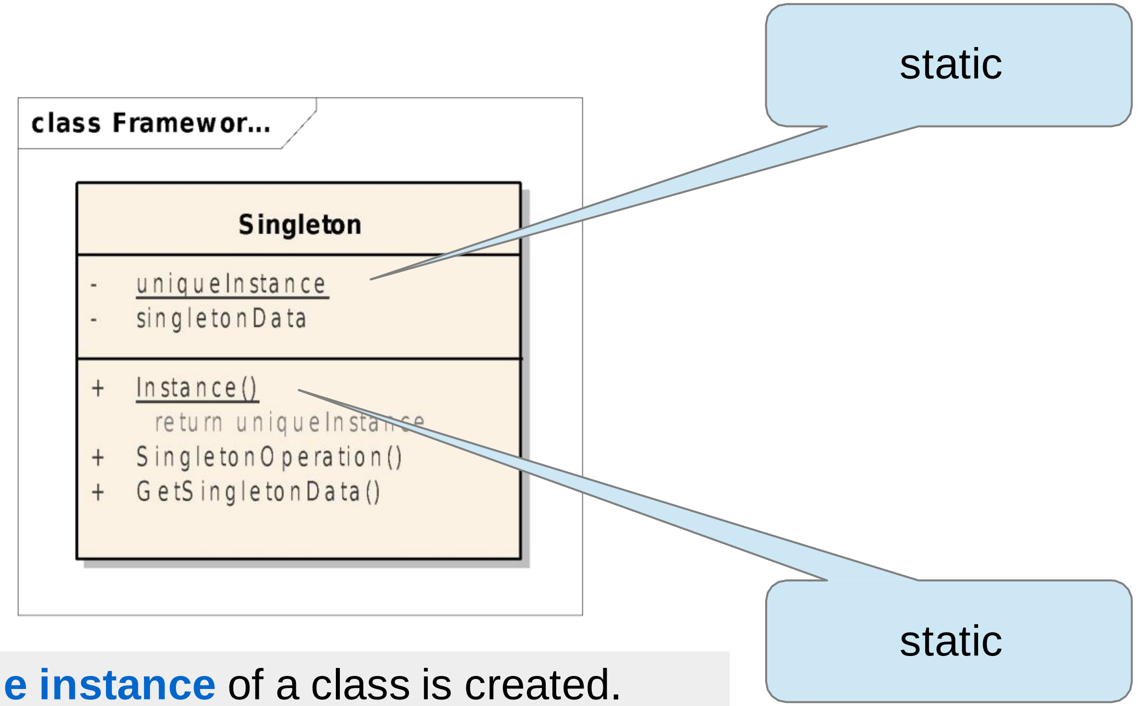
```
Start
Hello
Stop
```

Singleton Design Pattern

```
#include <string>
class Logger{
public:
    static Logger* Instance();
    bool openLogFile(std::string logFile);
    void writeToLogFile();
    bool closeLogFile();
private:
    Logger(){};  // Private so that it can  not be called
    Logger(Logger const&){}; // copy constructor is private
    Logger& operator=(Logger const&){};// assignment operator is private
    static Logger* m_pInstance;
};
```

http://www.yolinux.com/TUTORIALS/C++Singleton.html

Singleton Design Pattern



- Ensure that **only one instance** of a class is created.
- Provide a **global point of access** to the object.

Module 5

Object-Oriented Programming

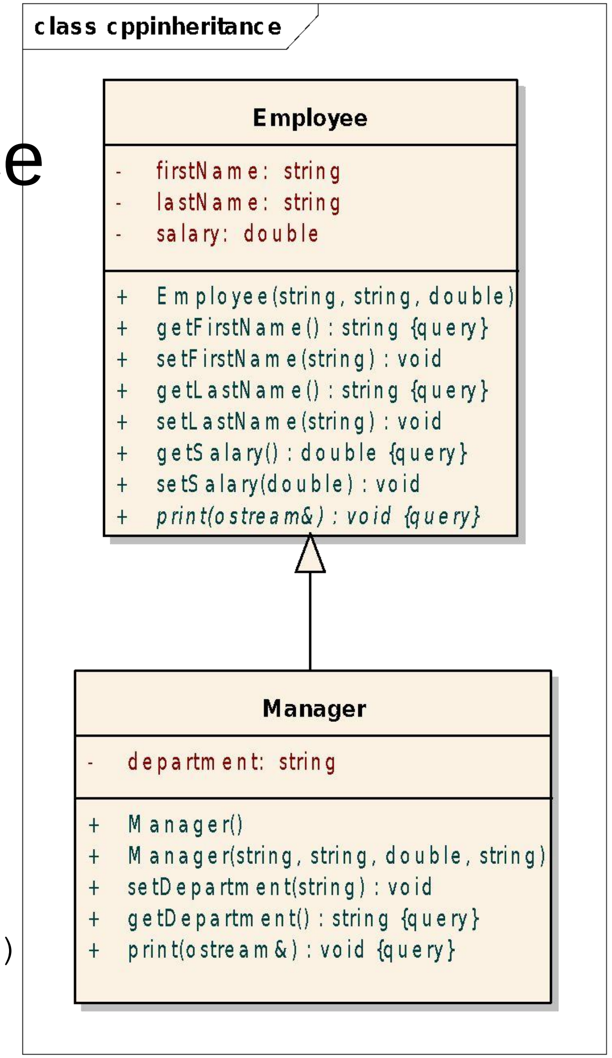
Public Inheritance

OOP: Inheritance

- Inheritance
  - *is-a* relationship - public inheritance
  - protected access
  - virtual member function
  - early (static) binding vs. late (dynamic) binding
  - abstract base classes
  - pure virtual functions
  - virtual destructor

OOP: Inheritance

- `public` inheritance
  - *is-a* relationship
  - base class: Employee
  - derived class: Manager
- You can do with inheritance
  - *add data*
    - ex. `department`
  - *add functionality*
    - ex. `getDepartment()`, `setDepartment()`
  - *modify methods' behavior*
    - ex. `print()`



OOP: Inheritance

- `protected` access
  - base class's private members can not be accessed in a derived class
  - base class's protected members can be accessed in a derived class
  - base class's public members can be accessed from anywhere

OOP: Inheritance

- `public` inheritance

```
class Employee{
public:
    Employee(string firstName = "", string lastName = "",
             double salary = 0.0) : firstName(firstName),
                                    lastName(lastName),
                                    salary(salary) {
    }
    //...
};
```

```
class Manager:public Employee{
    string department;
public:
    Manager();
    Manager( string firstName, string lastName, double salary,
             string department );
    //...
};
```

OOP: Inheritance

- Derived class's constructors

```
Manager::Manager(){
}
```

Employee's constructor invocation → Default constructor can be invoked implicitly

OOP: Inheritance

– Derived class's constructors

```
Manager::Manager(){
}
```

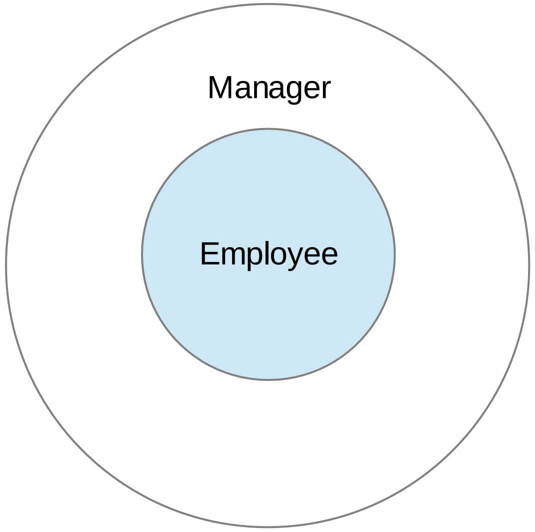
Employee's constructor invocation → Default constructor can be invoked implicitly

```
Manager::Manager(string firstName, string lastName, double salary,
                 string department):Employee(firstName, lastName, salary),
                                     artment(department){


}
```

base class's constructor invocation – *constructor initializer list*
arguments for the base class's constructor are specified in the definition of a derived class's constructor

OOP: Inheritance

- How are derived class's objects constructed?
  - *bottom up* order:
    - base class constructor invocation
    - member initialization
    - derived class's constructor block
  - destruction
    - in the opposite order



OOP: Inheritance

- Method overriding

```
class Employee{
public:
     virtual void print(ostream&) const;
};
```

```
class Manager:public Employee{
public:

    virtual void print(ostream&) const;
};
```

OOP: Inheritance

- Method overriding

```
class Employee {
public:
     virtual void print( ostream&) const;
};
```

```
void Employee::print(ostream& os ) const{
    os<<this->firstName<<" "<<this->lastName<<" "<<this->salary;
}
```

```
class Manager:public Employee{
public:
    virtual void print(ostream&) const;
};
```

```
void Manager::print(ostream& os) const{
    Employee::print(os);
    os<<" "<<department;
}
```

OOP: Inheritance

- Method overriding - virtual functions
  - non virtual functions are bound statically
    - compile time
  - virtual functions are bound dynamically
    - run time

OOP: Inheritance

- Polymorphism

```
void printAll( const vector<Employee*>& emps ){
    for( int i=0; i<emps.size(); ++i){
        emps[i]-> print(cout);
        cout<<endl;
    }
}

int main(int argc, char** argv) {
    vector<Employee*> v;
    Employee e("John", "Smith", 1000);
    v.push_back(&e);
    Manager m("Sarah", "Parker", 2000, "Sales");
    v.push_back(&m);
    cout<<endl;
    printAll( v );
    return 0;
}
```

Output:

```
John Smith 1000
Sarah Parker 2000 Sales
```

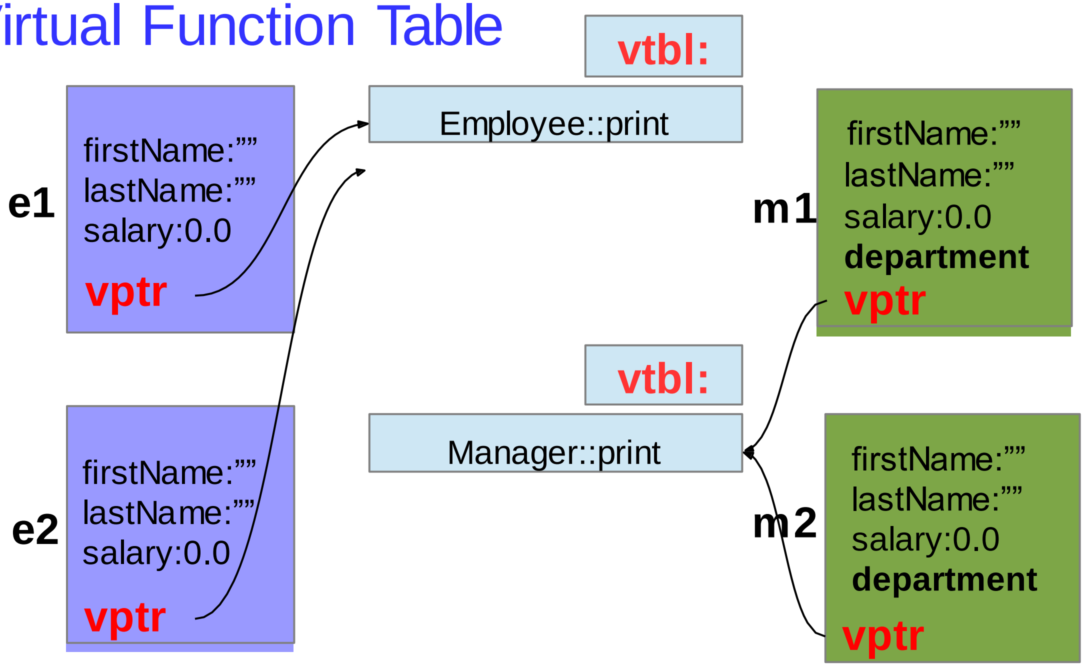
OOP: Inheritance

- Polymorphism
  - a type with virtual functions is called a polymorphic type
  - polymorphic behavior **preconditions**:
    - the member function must be virtual
    - objects must be manipulated through
      - pointers or
      - references
    - `Employee :: print( os )` static binding – no polymorphism

OOP: Inheritance

- Polymorphism – Virtual Function Table

```
class Employee{
public:
 virtual void print(ostream&) const;
      //...
};

class Manager:public Employee{
 virtual void print(ostream&) const;
      //...
};
Employee e1, e2;
Manager m1, m2;
```



Discussion!!!

```
Employee * pe;
pe = &e1; pe->print(); //???
pe = &m2; pe->print(); //???
```

Each class with virtual functions has its own virtual function table (vtbl).

RTTI – Run-Time Type Information

dynamic_cast<>(pointer)

```
class Base{};
class Derived : public Base{};

Base* basePointer = new Derived();
Derived* derivedPointer = nullptr;

//To find whether basePointer is pointing to Derived type of object

derivedPointer =
dynamic_cast<Derived*>(basePointer); if
(derivedPointer != nullptr){
   cout << "basePointer is pointing to a Derived class object";
}else{
   cout << "basePointer is NOT pointing to a Derived class object";
}
```

Java:
instanceof

RTTI – Run-Time Type Information

`dynamic_cast<>`(reference)

```
class Base{};
class Derived : public Base{};

Derived derived;
Base& baseRef = derived;

// If the operand of a dynamic_cast to a reference isn't of the expected type,
// a bad_cast exception is thrown.

try{
      Derived& derivedRef = dynamic_cast<Derived&>(baseRef);
} catch( bad_cast ){
      // ..
}
```

OOP: Inheritance

- Abstract classes
  - used for representing abstract concepts
  - used as base class for other classes
  - no instances can be created

OOP: Inheritance

- Abstract classes – pure virtual functions

```
class Shape{ // abstract class
    public:
        virtual void rotate(int)  = 0;      // pure virtual function
        virtual void draw()  = 0;  // pure virtual function
        // ...
};
```

```
 Shape s; //???
```

OOP: Inheritance

- Abstract classes – pure virtual functions

```
class Shape{ // abstract class
    public:
        virtual void rotate(int)  = 0;      // pure virtual function
        virtual void draw()  = 0;  // pure virtual function
        // ...
};
```

```
 Shape s; //Compiler error
```

OOP: Inheritance

- Abstract class → concrete class

```
class Point{ /* ... */ };
class Circle : public Shape {
    public:
        void rotate(int);          // override Shape::rotate
        void draw();               // override Shape::draw
        Circle(Point p, int r) ;
    private:
        Point center;
        int radius;
};
```

OOP: Inheritance

- Abstract class → abstract class

```
class Polygon : public Shape{
public:
    // draw() and rotate() are not overridden

};
```

```
Polygon p; //Compiler error
```

OOP: Inheritance

- Virtual destructor
  - Every class having at least one virtual function should have virtual destructor. Why?

```
class X{
public:
    // ...
    virtual ~X();
};
```

OOP: Inheritance

- Virtual destructor

```
void deleteAll( Employee ** emps, int size){
    for( int i=0; i<size; ++i){
        delete emps[ i ];
    }
    delete [] emps;
}

 // main
 Employee ** t = new Employee *[ 10 ];
 for(int i=0; i<10; ++i){
    if( i % 2 == 0 )
        t[ i ] = new Employee();
    else
        t[ i ] = new Manager();
 }
 deleteAll( t, 10);
```

Which destructor is invoked?

Module 6

Object-Oriented Programming

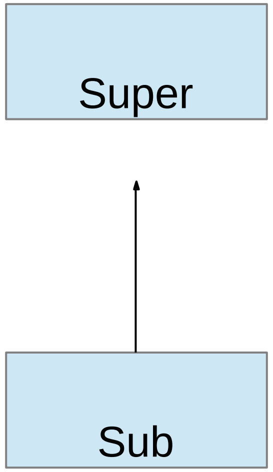
Object relationships

OOP: Object relationships

- The ***is-a*** relationship
  - Private inheritance
  - Multiple inheritance

- The ***has-a*** relationship
  - Association
    - Composition (strong containment)
    - Aggregation (weak containment)

OOP: Object relationships

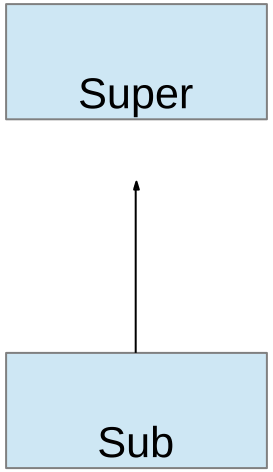


- The *is-a* relationship – *Client's view (1)*
  - works in only *one direction:*
    - every **Sub** object **is** also **a Super** one
    - but **Super** object **is not a Sub**

```
void foo1( const Super& s );
void foo2( const Sub& s);
Super super;
Sub sub;

foo1(super); //OK
foo1(sub);   //OK
foo2(super); //NOT OK
foo2(sub);   //OK
```

OOP: Object relationships

- The *is-a* relationship – *Client's view (2)*

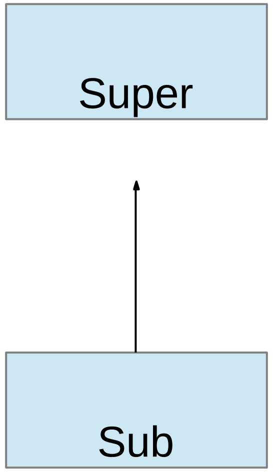



```
class Super{
public:
   virtual void method1();
};
class Sub : public Super{
public:
   virtual void method2();
};
```

```
Super * p= new Super();
p->method1(); //OK

p = new Sub();
p->method1(); //OK
p->method2(); //NOT OK
((Sub *)p)->method2();//OK
```

OOP: Object relationships

- The *is-a* relationship – *Sub-class's view*



- the `Sub` class augments the `Super` class by **adding additional methods**
- the `Sub` class **may override** the `Super` class **methods**
- the subclass can use all the public and protected members of a superclass.

OOP: Object relationships

- The *is-a* relationship: *preventing inheritance* ***C++11***
  - `final` classes – cannot be extended

```
class Super final
{

};
```

OOP: Object relationships

- The *is-a* relationship: *a client's view of overridden methods(1)*
  - *polymorphism*

```
class Super{
public:
    virtual void method1();
};
class Sub : public Super{
public:
    virtual void method1();
};
```

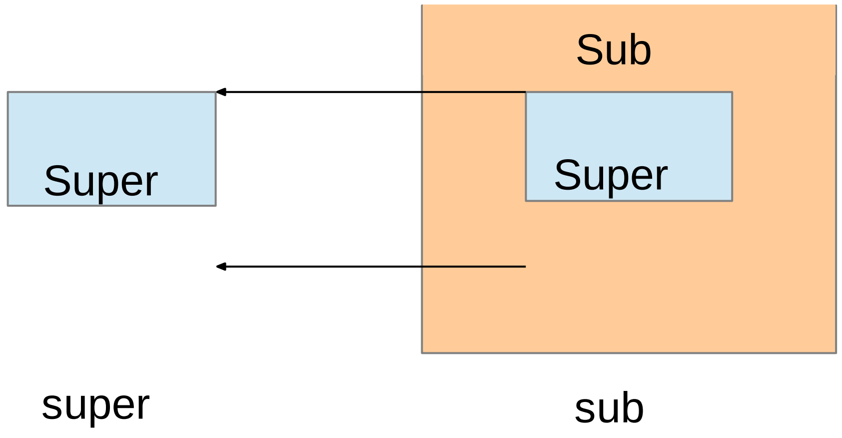
```
Super super;
super.method1(); //Super::method1()

Sub sub;
sub.method1();   //Sub::method1()

Super& ref =super;
ref.method1();   //Super::method1();

ref = sub;
ref.method1();   //Sub::method1();

Super* ptr =&super;
ptr->method1(); //Super::method1();

ptr = &sub;
ptr->method1();   //Sub::method1();
```

OOP: Object relationships

- The *is-a* relationship: *a client's view of overridden methods(2)*
  - object slicing

```
class Super{
public:
    virtual void method1();
};
class Sub : public Super{
public:
    virtual void method1();
};
```

```
Sub sub;
Super super = sub;
super.method1(); // Super::method1();
```



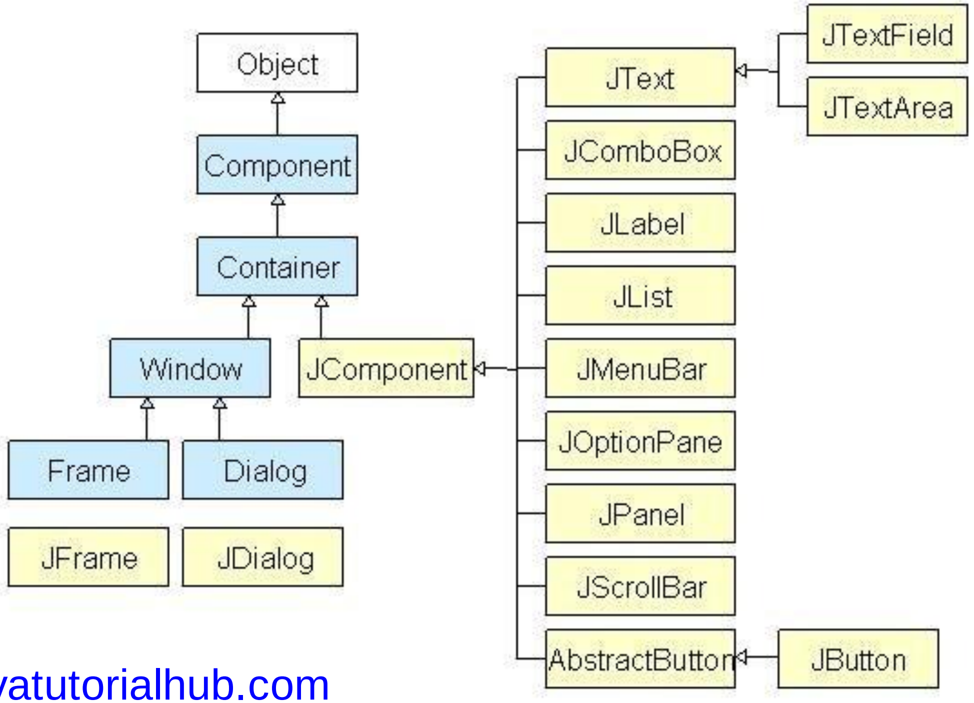
OOP: Object relationships

- The *is-a* relationship: *preventing method overriding* ***C++11***

```
class Super{
public:
    virtual void method1() final;
};
class Sub : public Super{
public:
    virtual void method1(); //ERROR
};
```

OOP: Object relationships

- Inheritance for polymorphism



www.javatutorialhub.com

OOP: Object relationships

- The *has-a* relationship

OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**

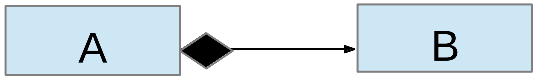
```
class B;

class
A{
private
:
    B b;
};
```

```
class B;

class
A{
private
:
    B*  b;
};
```

```
class B;

class
A{
private
:
    B&  b;
};
```

OOP: Object relationships

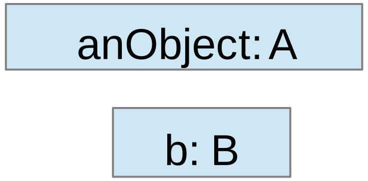
- Implementing the *has-a* relationship



  - An object **A** has an object **B**
    - **strong containment (composition)**

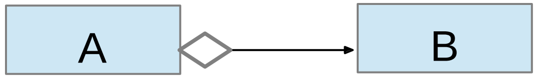
```
class B;

class A{
private:
    B b;
};
```

```
A anObject;
```



OOP: Object relationships

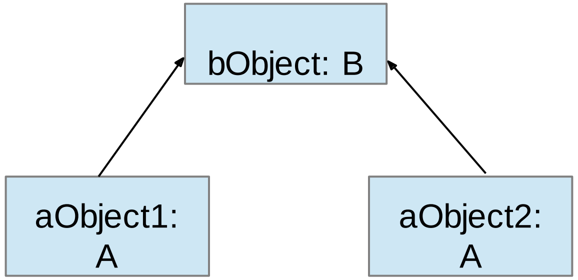
- Implementing the *has-a* relationship



  - An object **A** has an object **B**
    - **weak containment (aggregation)**

```
class B;

class A{
private:
    B& b;
public:
    A( const B& pb):b(pb){}
};
```

```
B bObject;
A aObject1(bObject);
A aObject2(bObject);
```



OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**

weak containment

```
class B;

class A{
private:
     B* b;
public:
     A( B* pb):b( pb ){}
};
```

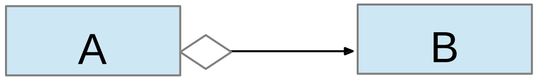
strong containment

```
class B;

class A{
private:
     B* b;
public:
     A(){
          b = new B();
     }
     ~A(){
          delete b;
     }
};
```

OOP: Object relationships

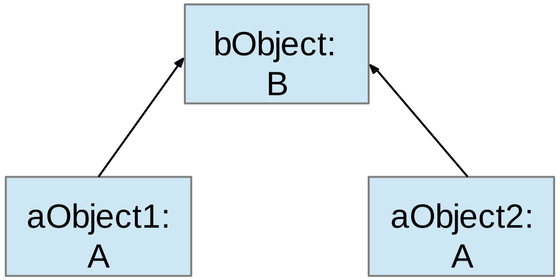
- Implementing the *has-a* relationship
  - An object **A** has an object **B**

weak containment

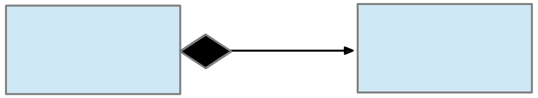



```
class B;

class A{
private:
     B* b;
public:
     A( B* pb):b( pb ){}
};
```

```
Usage:
     B bObject;
     A aObject1(&bObject);
     A aObject2(&bObject);
```



OOP: Object relationships



– Implementing the *has-a* relationship

A

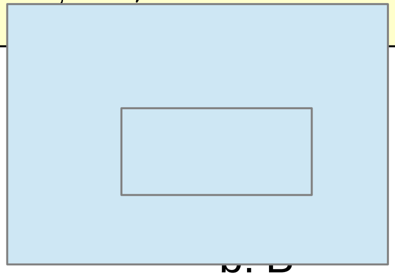
B

- An object **A** has an object

B strong containment

```
class B;

class
A{
priva
te:
    B*
b;
public
:
    A(){
        b = new B();
    }
    ~A(){
        delete b;
```

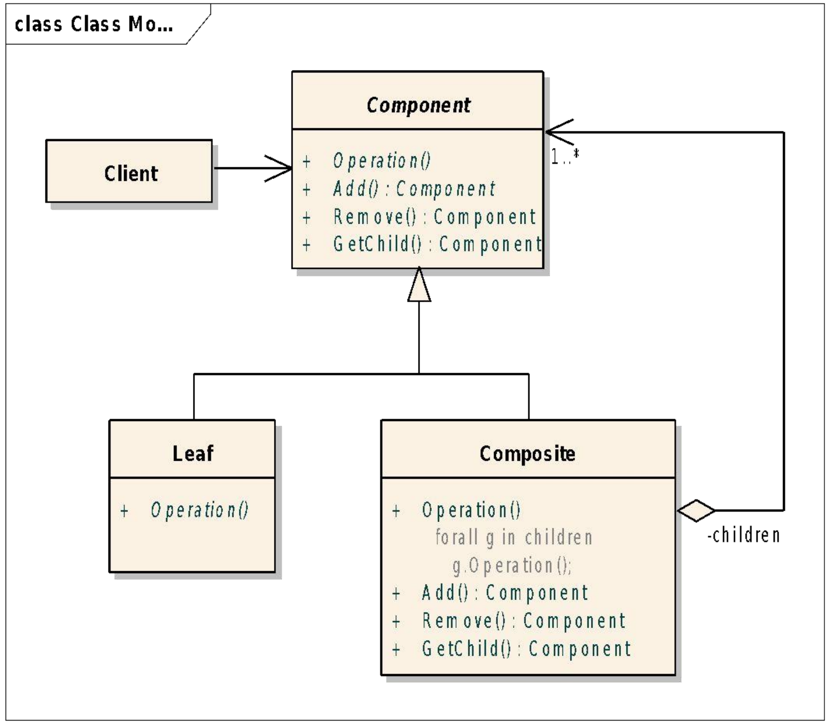
```
Usage:
    A aObject;
```



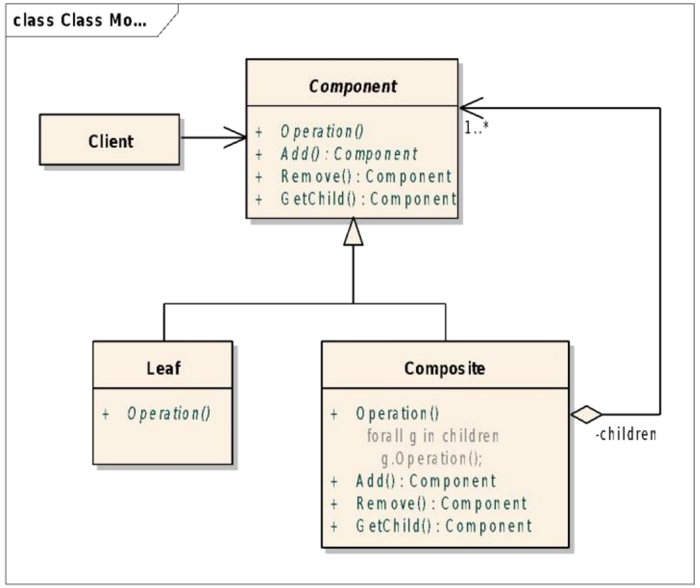
b: B

OOP: Object relationships

- Combining the *is-a* and the *has-a* relationships



Composite Design Pattern



- Compose objects into tree structures to represent **part-whole hierarchies.**
- Lets clients treat **individual objects** and **composition of objects uniformly.**

Composite Design Pattern

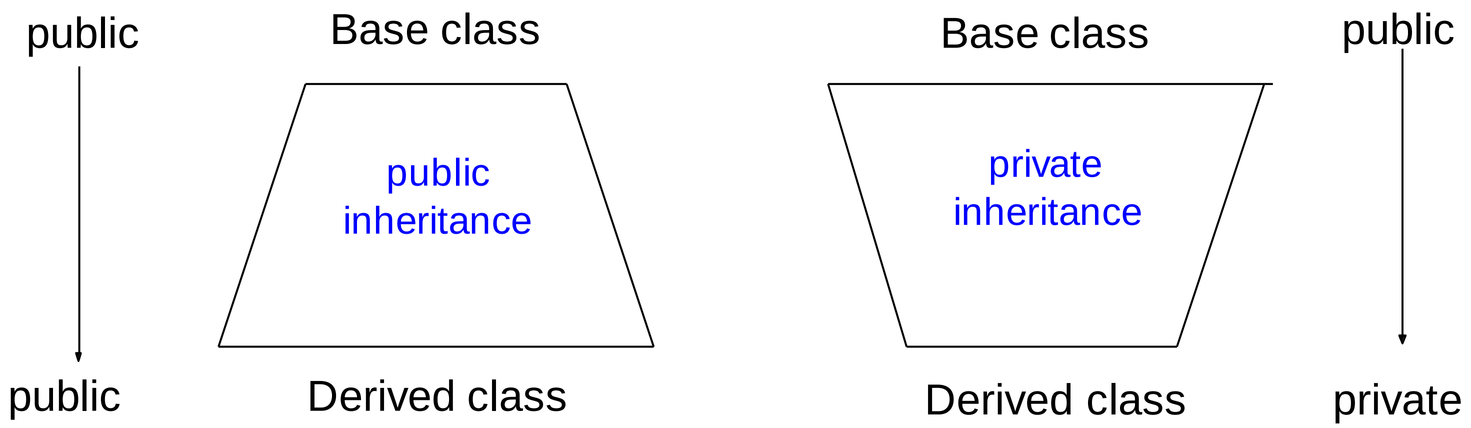
Examples:

- Menu – MenuItem: Menus that contain menu items, each of which could be a menu.
- Container – Element: Containers that contain Elements, each of which could be a Container.
- GUI Container – GUI component: GUI containers that contain GUI components, each of which could be a container

Source: http://www.oodesign.com/composite-pattern.html

Private Inheritance

– another possibility for *has-a* relationship



Derived class **inherits** the base class behavior

Derived class **hides** the base class behavior

Private Inheritance

```
template <typename T>
class MyStack : private vector<T> {
public:
    void push(T elem) {
        this->push_back(elem);
    }
    bool isEmpty() {
        return this->empty();
    }
    void pop() {
        if (!this->empty())this->pop_back();
    }
    T top() {
        if (this->empty()) throw out_of_range("Stack is empty");
        else return this->back();
    }
};
```

Why is **public inheritance** in this case dangerous???

Non-public Inheritance

- it is very rare;
- use it cautiously;
- most programmers are not familiar with it;

What does it print?

```
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
            cout<<"Super"<<endl;
    }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d){
            cout<<"Sub"<<endl;
    }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

What does it print?

```
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
            cout<<"Super"<<endl;
     }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d){
            cout<<"Sub"<<endl;
     }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

creates a new method, instead of overriding the method

The `override` keyword C++11

```
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
            cout<<"Super"<<endl;
     }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d) const override{
            cout<<"Sub"<<endl;
     }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

Object-Oriented Programming (OOP)

Content

- Classes and Objects
- Advanced Class Features
- Operator overloading
- Object Relationships
- Abstraction

OOP: Classes and Objects

Content

- Members of the class. Access levels. Encapsulation.
- Class: **interface + implementation**
- Constructors and **destructors**
- **`const`** member functions
- Constructor initializer
- Copy constructor
- Object's lifecycle

OOP: Types of Classes

Types of classes:

– **Polymorphic** Classes – *designed for extension*
  - Shape, exception, ...
– **Value** Classes – *designed for storing values*
  - int, complex<double>, ...
– **RAII** (**R**esource **A**cquisition **I**s **I**nitialization) Classes –
– (encapsulate a **resource** into a class → resource lifetime object lifetime)
  - thread, unique_ptr, ...

What type of resource?

OOP: Classes and objects

Class = Type (Data + Operations)

- Members of the class
- **Data:**
  - data members (properties, attributes)
- **Operations:**
  - methods (behaviors)
- Each member is associated with an **access level**:
  - `private` **-**
  - `public` **+**
  - `protected` **#**

OOP: Classes and objects

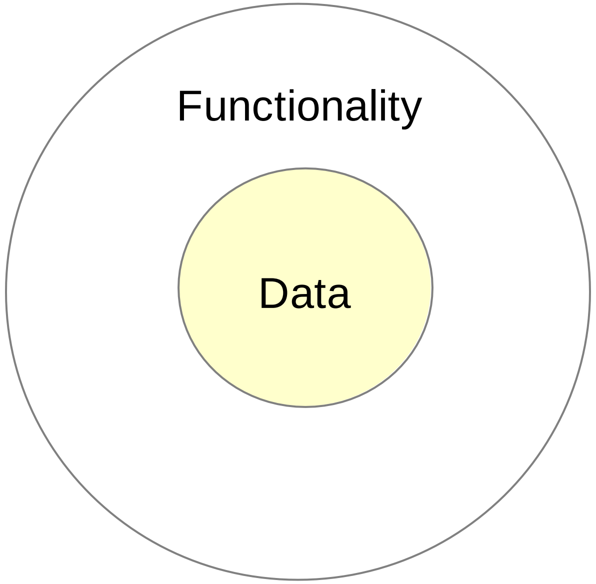
Object = Instance of a class

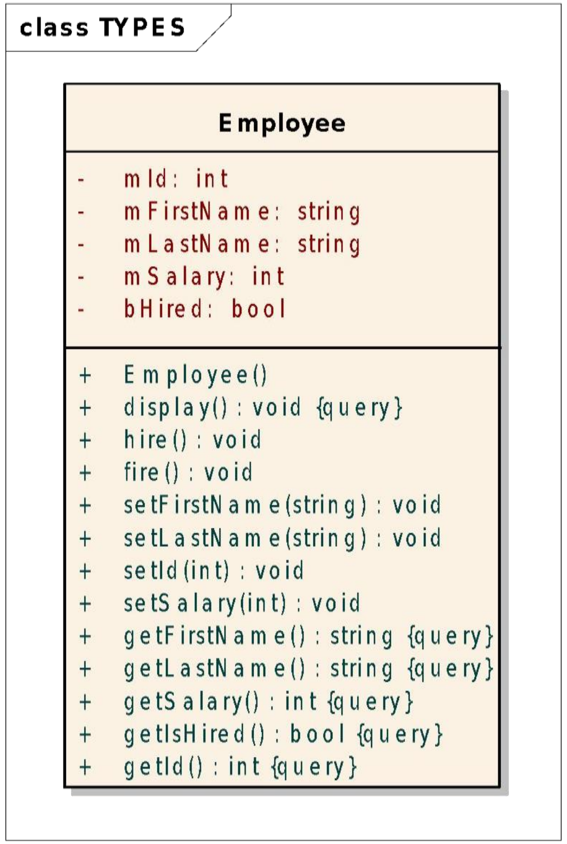
- An employee object: `Employee emp;`
  - **Properties** are the characteristics that describe an object.
    - *What makes this object different?*
      - `id, firstName, lastName, salary, hired`
  - **Behaviors** answer the question:
    - *What can we do to this object?*
      - `hire(), fire(), display()`, `get` and `set` data members

OOP: Classes and objects

Encapsulation

- an object encapsulates *data* and *functionality*.





OOP: Classes and objects

Class creation

- class **declaration** - *interface*
  - `Employee.h`
- class **definition** – *implementation*
  - `Employee.cpp`

OOP: Classes and objects

Employee.h

```
class Employee{
public:
    Employee();
    void display() const;
    void hire();
    void fire();
    // Getters and setters
    void setFirstName( string inFirstName );
    void setLastName ( string inLastName );
    void setId( int inId );
    void setSalary( int inSalary );
    string getFirstName() const;
    string getLastName() const;
    int getSalary() const;
    bool getIsHired() const;
     int getId() const;
private:
    int mId;
    string mFirstName;
    string mLastName;
    int mSalary;
    bool bHired;
};
```

Methods' declaration

Data members

OOP: Classes and objects

The Constructor and the object's state

- The **state of an object** is defined by its data members.
- The **constructor** is responsible for the **initial state** of the object

```
Employee :: Employee() : mId(-1),
                         mFirstName(""),
                         mLastName(""),
                         mSalary(0),
                         bHired(false){
}
```

Members are initialized through the constructor initializer list

```
Employee :: Employee() {
        mId = -1;
        mFirstName="";
        mLastName="";
        mSalary =0;
        bHired = false;
}
```

Members are assigned

Only constructors can use this initializer-list syntax!!!

OOP: Classes and objects

Constructors

- *responsibility:* data members initialization of a class object
- invoked automatically for each object
- have the *same name* as the class
- have *no return type*
- a class can have *multiple constructors* (function **overloading**)
- may not be declared as `const`
  - constructors can write to const objects

OOP: Classes and objects

Member initialization (C++11)

```
class C{
    string s ("abc");
    double d = 0;
    char * p {nullptr};
    int y[4] {1,2,3,4};
public:
    C(){}
};
```

Compiler

```
class C{
   string s;
   double d;
   char * p;
   int y[5];
public:
   C():s("abc"),
   d(0.0),p(nullptr),
   y{1,2,3,4} {}
};
```

OOP: Classes and objects

Defining a member function

- Employee.cpp
- A `const` **member function** cannot change the object's state, can be invoked on const objects

```
void Employee::hire(){
    bHired = true;
}

string Employee::getFirstName() const{
    return mFirstName;
}
```

OOP: Classes and objects

Defining a member function

```
void Employee::display() const {
     cou  << "Employee: " << getLastName() << ",
     t    << "
          << getFirstName() << endl;
     cou     "-------------------------" << endl;
     t
    cout << (bHired ? "Current Employee" :
                       "Former Employee") <<
                       endl;
    cout << "Employee ID: " << getId() << endl;
    cout << "Salary: " << getSalary() << endl;
    cout << endl;
}
```

OOP: Classes and objects

TestEmployee.cpp

- Using `const` member functions

```
void foo( const Employee& e){
   e.display();  // OK. display() is a const member function
   e.fire();     // ERROR. fire() is not a const member function
}

int main() {
    Employee emp;
    emp.setFirstName("Robert");
    emp.setLastName("Black");
    emp.setId(1);
    emp.setSalary(1000);
    emp.hire();
    emp.display();
    foo( emp );
    return 0;
}
```

OOP: Classes and objects

Interface: **Employee.h**

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

class Employee{
public:
    Employee();
    //...
protected:
    int mId;
    string mFirstName;
    string mLastName;
    int mSalary;
    bool bHired;
};

#endif
```

Implementation: **Employee.cpp**

```
#include "Employee.h"


Employee::Employee() :
  mId(-1),
  mFirstName(""),
  mLastName(""),
  mSalary(0),
  bHired(false){
}

string Employee::getFirstName() const{
    return mFirstName;
}
/
/ ...
```

OOP: Classes and objects

Object life cycles:

- creation
- assignment
- destruction

OOP: Classes and objects

Object creation:

```
int main() {
    Employee emp;
    emp.display();

    Employee *demp = new Employee();
    demp->display();
    // ..
    delete demp;
    return 0;
}
```

object's lifecycle

- all its *embedded objects* are also created

OOP: Classes and objects

Object creation – constructors:

- *default constructor* (0-argument constructor)

```
Employee :: Employee() : mId(-1), mFirstName(""),
mLastName(""), mSalary(0), bHired(false){
}
```

```
Employee :: Employee() {
}
```

- `Employee employees[ 10 ];`
- `vector<Employee>  emps(10);`

- memory allocation
- constructor call on each allocated object

OOP: Classes and objects

Object creation – constructors:

- *Compiler-generated default constructor*

- if a class *does not specify* any constructors, the *compiler will generate* one that does not take any arguments

```
class Value{
public:
    void setValue( double inValue);
    double getValue() const;
private:
    double value;
};
```

OOP: Classes and objects

Constructors: `default` and `delete` specifiers (C++ 11)

```
class X{
    int i = 4;
    int j {5};
public:
    X(int a) : i{a} {} // i = a, j = 5
    X() = default;       // i = 4, j = 5
};
```

Explicitly forcing the automatic generation of a **default** constructor by the compiler.

OOP: Classes and objects

Constructors: **`default`** and **`delete`** specifiers (C++ 11)

```
class X{
public:
    X( double ){}
};

X x2(3.14); //OK
X x1(10); //OK
```

int → double conversion

```
class X{
public:
    X( int )= delete;
    X( double );
};
X x1(10); //ERROR
X x2(3.14); //OK
```

OOP: Classes and objects

Best practice: *always provide default values for members!* C++ 11

```
struct Point{
int x, y;
    Point ( int x = 0, int y = 0 ): x(x), y(y){}
};
class Foo{
    int i {};
    double d {};
    char c {};
    Point p {};
public:
    void print(){
        cout <<"i: "<<i<<endl; cout <<"d: "<<d<<endl;
        cout <<"c: "<<c<<endl;
        cout <<"p: "<<p.x<<", "<<p.y<<endl;
```

```
int main() {
    Foo f;
    f.print();
    return
}
```

```
p: 0, 0
```

OOP: Classes and objects

Constructor initializer

```
class ConstRef{
public:
    ConstRef( int& );
private:
    int mI;
    const int mCi;
    int& mRi;
};

ConstRef::ConstRef( int& inI ){
    mI = inI;   //OK
    mCi = inI; //ERROR: cannot assign to a const
    mRi = inI; //ERROR: uninitialized reference member
}
```

```
ConstRef::ConstRef( int& inI ): mI( inI ), mCi( inI ), mRi( inI ){}
```

ctor initializer

OOP: Classes and objects

Constructor initializer

- data types that must be initialized in a ctor-initializer
  - `const` data members
  - reference data members
  - object data members having no default constructor
  - superclasses without default constructor

OOP: Classes and objects

A *non-default* Constructor

```
Employee :: Employee( int inId, string inFirstName,
                      string inLastName,
                      int inSalary, int inHired) :
          mId(inId), mFirstName(inFirstName),
          mLastName(inLastName), mSalary(inSalary),
          bHired(inHired){
}
```

OOP: Classes and objects

Delegating Constructor (C++11)

```
class SomeType{
  int number;

public:
  SomeType(int newNumber) : number(newNumber) {}
  SomeType() : SomeType(42) {}
};
```

OOP: Classes and objects

Copy Constructor

```
Employee emp1(1, "Robert", "Black", 4000, true);
```

- called in one of the following cases:
  - `Employee emp2( emp1 ); //copy-constructor called`
  - `Employee emp3 = emp2;  //copy-constructor called`
  - `void foo( Employee emp );//copy-constructor called`
- if you don't define a copy-constructor explicitly, the compiler creates one for you
  - this performs a **bitwise** copy

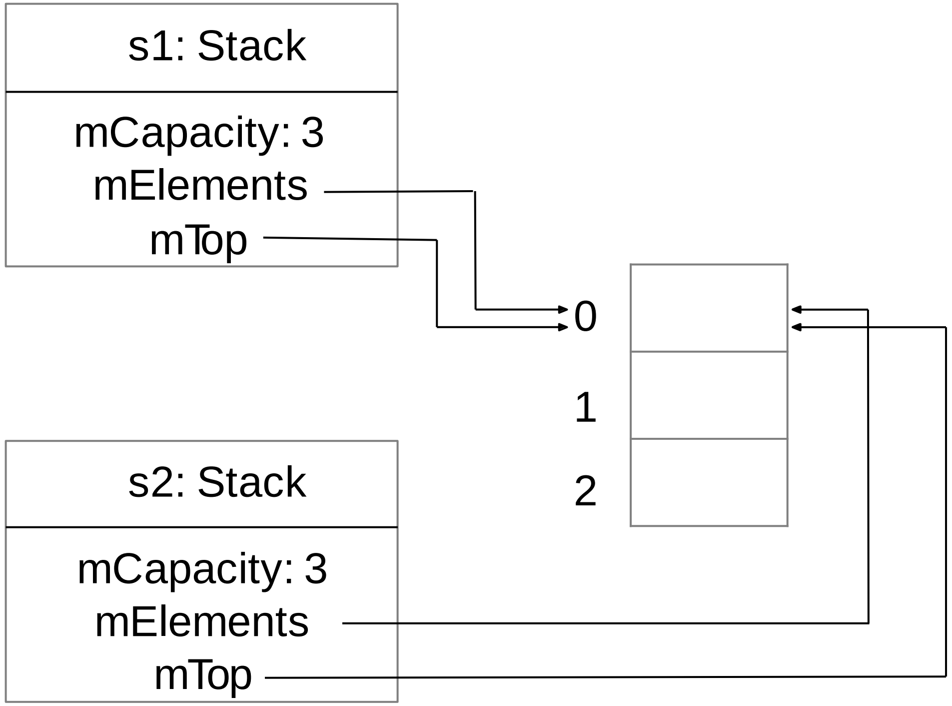
OOP: Classes and objects

```
//Stack.h

#ifndef STACK_H
#define  STACK_H

class Stack{
public:
    Stack( int inCapacity );
    void push( double inDouble );
    double top() const;
    void pop();
    bool isFull() const;
    bool isEmpty()const;

private:
    int mCapacity;
    double * mElements;
    double * mTop;
};
#endif   /* STACK_H */
```

```
//Stack.cpp

#include  "Stack.h"

 Stack::Stack( int inCapacity ){
     mCapacity = inCapacity;
     mElements = new double [ mCapacity ];
     mTop = mElements;
 }

 void Stack::push( double inDouble ){
     if( !isFull()){
         *mTop = inDouble;
         mTop++;
     }
 }
```

OOP: Classes and objects

```
//TestStack.cpp
#include "Stack.h"

int main(){
      Stack s1(3);
     Stack s2 = s1;
     s1.push(1);
     s2.push(2);

     cout<<"s1: "<<s1.top()<<endl;
     cout<<"s2: "<<s2.top()<<endl;
}
```



OOP: Classes and objects

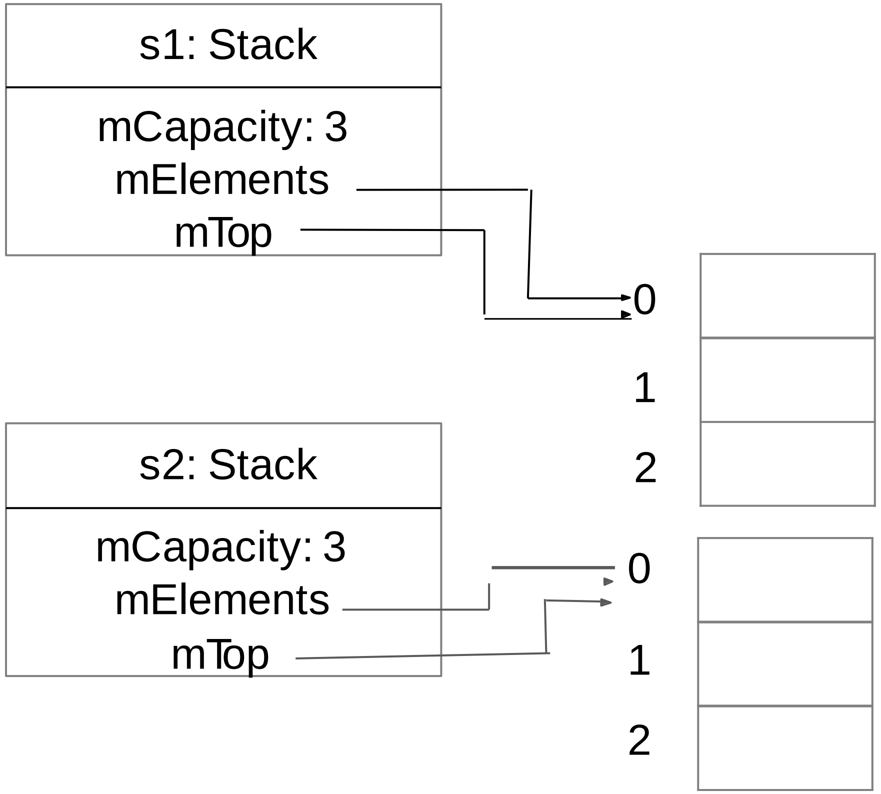
Copy constructor: **T (const T&)**

```
//Stack.h

#ifndef STACK_H
#define  STACK_H

class Stack{
public:
    //Copy constructor
    Stack( const Stack& );
private:
    int mCapacity;
    double * mElements;
    double * mTop;
};
#endif   /* STACK_H */
```

```
//Stack.cpp

#include "Stack.h"

Stack::Stack( const Stack& s ){
     mCapacity = s.mCapacity;
     mElements = new double[ mCapacity ];
     int nr = s.mTop - s.mElements;
     for( int i=0; i<nr; ++i ){
        mElements[ i ] = s.mElements[ i ];
     }
     mTop = mElements + nr;
 }
```

OOP: Classes and objects

```
//TestStack.cpp
#include "Stack.h"

int main(){
     Stack s1(3);
     Stack s2 = s1;
     s1.push(1);
     s2.push(2);

     cout<<"s1: "<<s1.top()<<endl;
     cout<<"s2: "<<s2.top()<<endl;
}
```



OOP: Classes and objects

Destructor

- when an object is destroyed:
  - the object's destructor is automatically invoked,
  - the memory used by the object is freed.
- each class has one destructor
- usually place to perform cleanup work for the object
- if you don't declare a destructor → the compiler will generate one, which destroys the object's member

OOP: Classes and objects

Destructor

– Syntax: `T :: ~T();`

```
Stack::~Stack(){
    if( mElements != nullptr ){
        delete[] mElements;
        mElements = nullptr;
    }
}
```

```
{    // block begin
     Stack s(10);          // s: constructor
    Stack* s1 = new Stack(5);// s1: constructor
    s.push(3);
     s1->push(10);
     delete s1;               //s1: destructor
     s.push(16);
}   // block end              //s: destructor
```

OOP: Classes and objects

Default parameters

- if the user specifies the arguments → the defaults are ignored
- if the user omits the arguments → the defaults are used
- the default parameters are specified **only** in the ***method declaration*** (not in the definition)

```
//Stack.h
class Stack{
public:
    Stack( int inCapacity = 5 );
    ..
};
//Stack.cpp
Stack::Stack( int inCapacity ){
     mCapacity = inCapacity;
     mElements = new double [ mCapacity ];
     mTop = mElements;
}
```

```
//TestStack.cpp

Stack  s1(3);  //capacity:  3
Stack  s2;     //capacity:  5
Stack s3( 10 ); //capacity: 10
```

OOP: Classes and objects

The `this` pointer

- every method call passes a pointer to the object for which it is called as *hidden parameter* having the name `this`
- Usage:
  - for disambiguation

```
Stack::Stack( int mCapacity ){
    this → mCapacity = mCapacity;
    //..
}
```

OOP: Classes and objects

Programming task [Prata]

```
class Queue
{
   enum {Q_SIZE = 10};
private:
   // private representation to be developed later
public:
   Queue(int qs = Q_SIZE); // create queue with a qs limit
   ~Queue();
   bool isempty() const;
   bool isfull()  const;
   int queuecount() const;
   bool enqueue(const Item &item); // add item to end
   bool dequeue(Item &item); // remove item from front
};
```

OOP: Classes and objects

Programming task [Prata]

```
class Queue
{
private:
    // class scope definitions

    // Node is a nested structure definition local to this class
    struct Node { Item item; struct Node * next;};
    enum {Q_SIZE = 10};

    // private class members
    Node * front; // pointer to front of Queue
    Node * rear; // pointer to rear of Queue
    int items; // current number of items in Queue
    const int qsize; // maximum number of items in Queue

};
```

Module 3

Object-Oriented Programming

Advanced Class Features

OOP: Advanced class features

Content

- Inline functions
- Stack vs. Heap
- Array of objects vs. array of pointers
- Passing function arguments
- Static members
- Friend functions, friend classes
- Nested classes
- Move semantics (C++11)

OOP: Advanced class features

`Inline` functions

- designed to speed up programs (like macros)
- the compiler replaces the function call with the function code (no function call!)
- advantage: *speed*
- disadvantage: *code bloat*
    - ex. 10 function calls → 10 * function's size

OOP: Advanced class features

How to make a function `inline`?

- use the `inline` keyword either in function declaration or in function definition
- both member and standalone functions can be `inline`
- common practice:
  - place the implementation of the `inline` function into the header file
- only small functions are eligible as `inline`
- the compiler may completely ignore your request

OOP: Advanced class features

`inline` function examples

```
inline double square(double a){
   return a * a;
}

class Value{
   int value;
public:
   inline int getValue()const{ return value; }

    inline void setValue( int value ){
       this->value = value;
   }
};
```

OOP: Advanced class features

- Stack vs. Heap
- Heap – Dynamic allocation

```
void draw(){
    Point * p = new Point();
    p->move(3,3);
    //...
    delete p;
}
```

- Stack – Automatic allocation

```
void draw(){
    Point p;
    p.move(6,6);
    //...
}
```

OOP: Advanced class features

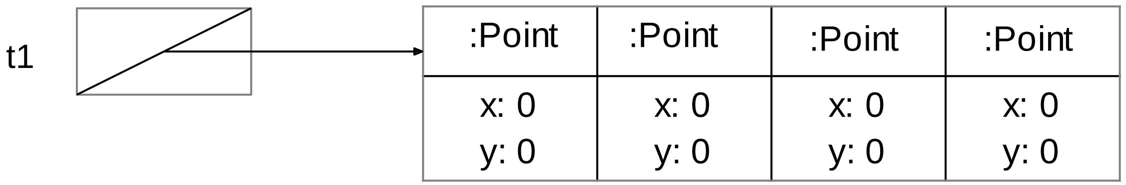
Array of objects

```
class Point{
     int x, y;
public:
     Point( int x=0, int y=0);
     //...
};
```

What is the difference between these two arrays?

Point * t1 = new Point[4];

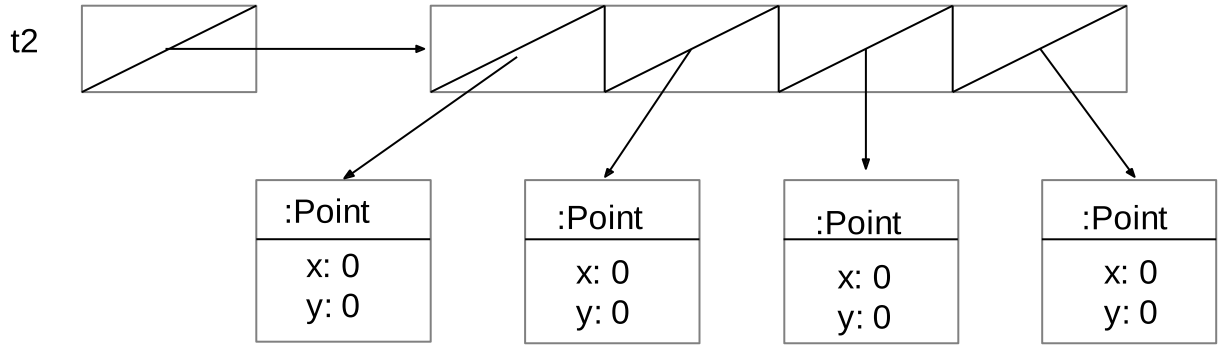
Point t1[4];



OOP: Advanced class features

Array of pointers

```
Point ** t2 = new Point*[ 4 ];
for(int i=0; i<4; ++i ){
     t2[i] = new Point(0,0);
}
for( int i=0; i<4; ++i ){
     cout<<*t2[ i ]<<endl;
}
```



OOP: Advanced class features

Static members:

- `static` methods
- `static` data

- Functions belonging to a *class scope* which don't access object's data can be `static`
- Static methods can't be `const` methods (they do not access object's state)
- They are not called on specific objects ⇒ they have no `this` pointer

OOP: Advanced class features

- Static members

```
//Complex.h

class Complex{
public:
    Complex(int re=0, int im=0);
    static int getNumComplex();
    // ...
private:
   static int num_complex;
   double re, im;
};
```

instance counter

initializing static class member

```
//Complex.cpp

int Complex::num_complex = 0;

int Complex::getNumComplex(){
    return num_complex;
}

Complex::Complex(int re, int im){
    this->re = re;
    this->im = im;
    ++num_complex;
}
```

OOP: Advanced class features

- Static method invocation

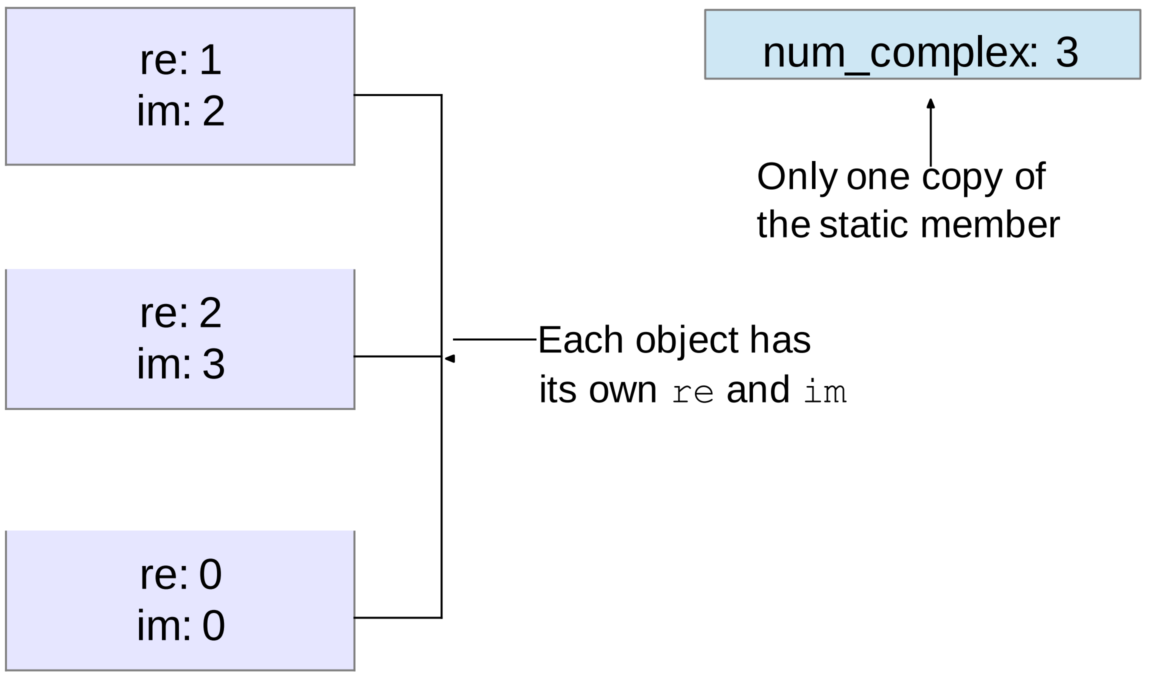
elegant

```
Complex z1(1,2), z2(2,3), z3;
cout<<"Number of complexs:"<<Complex::getNumComplex()<<endl;

 cout<<"Number of complexes: "<<z1.getNumComplex()<<endl;
```

non - elegant

OOP: Advanced class features

Complex z1(1,2), z2(2,3), z3;



OOP: Advanced class features

- Classes vs. Structs
  - default access specifier
    - class: `private`
    - struct: `public`
  - class: data + methods, can be used polymorphically
  - struct: mostly data + convenience methods

OOP: Advanced class features

– Classes vs. structures

```
Class list{
private:
  struct node
  {
    node *next;
    int val;
    node( int val = 0, node * next = nullptr):val(val), next(next){}
  };
  node * mHead;
public:
  list ();
  ~list ();
  void insert (int a);
  void printAll()const;
};
```

OOP: Advanced class features

- Passing function arguments
  - by value
    - the function works on a copy of the variable
  - by reference
    - the function works on the original variable, may modify it
  - by constant reference
    - the function works on the original variable, may not modify (verified by the compiler)

OOP: Advanced class features

- Passing function arguments

passing primitive values

```
void f1(int x)  {x = x + 1;}
void f2(int& x) {x = x + 1;}
void f3(const int& x) {x = x + 1;}// !!!!
void f4(int *x) {*x = *x + 1;}

int main(){
    int y = 5;
    f1(y);
    f2(y);
    f3(y);
    f4(&y);
    return 0;
}
```

OOP: Advanced class features

– Passing function arguments

```
void f1(Point p);
void f2(Point& p);
void f3(const Point& p);
void f4(Point *p);

int main(){
    Point p1(3,3);
    f1(p1);
    f2(p1);
    f3(p1);
    f4(&p1);
    return 0;
}
```

copy constructor will be used on the argument

only `const` methods of the class can be invoked on this argument

OOP: Advanced class features

- `friend` functions, `friend` classes, `friend` member functions
  - friends are allowed to access private members of a class
  - Use it rarely
    - operator overloading

OOP: Advanced class features

- `friend` vs. `static` functions

```
class Test{
private:
    int iValue;
    static int sValue;
public:
    Test( int in ):iValue( in ){}
    void print() const;
    static void print( const Test& what );
    friend void print( const Test& what );
};
```

OOP: Advanced class features

- `friend` vs. `static` functions

```
int Test :: sValue = 0;

void Test::print() const{
    cout<<"Member: "<<iValue<<endl;
}

void Test::print( const Test& what ){
    cout<<"Static: "<<what.iValue<<endl;
}

void print( const Test& what ){
    cout<<"Friend: "<<what.iValue<<endl;
}

int main() {
    Test test( 10 );
    test.print();
    Test::print( test );
    print( test );
}
```

OOP: Advanced class features

- `friend` class vs. `friend` member function

```
class List{
private:
     ListElement * head;
public:
     bool find( int key );
     …
};
```

```
class ListElement{
private:
     int key;
     ListElement * next;
     friend class List;
     ...
};
```

```
Class ListElement{
private:
     int key;
     ListElement * next;
     friend class List::find( int key);
     ...
};
```

OOP: Advanced class features

C++03

- Returning a reference to a `const` object

```
// version 1
vector<int> Max(const vector<int> & v1, const vector<int> & v2){
    if (v1.size() > v2.size())
        return v1;
    else
        return v2;
}

// version 2
const vector<int> & Max(const vector<int> & v1, const vector<int> & v2){
    if (v1.size() > v2.size())
        return v1;
    else
        return v2;
}
```

Copy constructor invocation

More efficient

The reference should be to a non-local object

OOP: Advanced class features

C++11

- Returning a reference to a `const` object

```
vector<int> selectOdd( const vector<int>& v){
   vector<int> odds;
   for( int a: v ){
       if (a % 2 == 1 ){
           odds.push_back( a );
       }
   }
   return odds;
}

//...
vector<int> v(N);
for( int i=0; i<N; ++i){
    v.push_back( rand()% M);
}
vector<int> result = selectOdd( v );
```

EFFICIENT!
MOVE
constructor
invocation

OOP: Advanced class features

- Nested classes
  - the class declared within another class is called a *nested class*
  - usually helper classes are declared as nested

```
// Version 1

class Queue
{
 private:
     // class scope definitions
     // Node is a nested structure definition local to this class
     struct Node {Item item; struct Node * next;};
     ...
};
```

OOP: Advanced class features

- Nested classes [Prata]

Node visibility!!!

```
// Version 2

class Queue
{
     // class scope definitions
     // Node is a nested class definition  local to this class
     class Node
     {
     public:
          Item item;
          Node * next;
          Node(const Item & i) : item(i), next(0) { }
     };
     //...
};
```

OOP: Advanced class features

- Nested classes
  - a nested class **B** declared in a **private** section of a class **A**:
    - **B** is local to class **A** (only class A can use it)
  - a nested class **B** declared in a **protected** section of a class **A**:
    - **B** can be used both in **A** and in the derived classes of **A**
  - a nested class **B** declared in a **public** section of a class **A**:
    - **B** is available to the outside world (Usage: `A::B b;`)

OOP: Advanced class features

- Features of a *well-behaved* C++ class
  - implicit constructor
    - `T :: T(){ … }`
  - destructor
    - `T :: ~T(){ … }`
  - copy constructor
    - `T :: T( const T& ){ … }`
  - assignment operator (*see next module*)
    - `T& T :: operator=( const T& ){ … }`

OOP: Advanced class features

- Constructor delegation (C++11)

```
// C++03
class A
{
    void init() { std::cout << "init()"; }
    void doSomethingElse() { std::cout << "doSomethingElse()\n"; }
public:
    A() { init(); }
    A(int a) { init(); doSomethingElse(); }
};
```

```
// C++11
class A
{
    void doSomethingElse() { std::cout << "doSomethingElse()\n"; }
public:
    A() { ... }
    A(int a) : A() { doSomethingElse(); }
};
```

OOP: Advanced class features

- Lvalues:
  - Refer to objects accessible at more than one point in a source code
    - Named objects
    - Objects accessible via pointers/references
  - Lvalues may not be moved from
- Rvalues:
  - Refer to objects accessible at exactly one point in source code
    - Temporary objects (e.g. by value function return)
  - Rvalues may be moved from

OOP: Advanced class features

- Lvalue

```
int  x;
x = 10;
```

Rvalue

OOP: Advanced class features

- Move Semantics (C++11)

```
class string{
    char* data;
public:
   string( const char* );
   string( const string& );
   ~string();
};
```

```
string :: string(const char* p){
    size_t size = strlen(p) + 1;
    data = new char[size];
    memcpy(data, p, size);
}
string :: string(const string& that){
    size_t size = strlen(that.data) + 1;
    data = new char[size];
    memcpy(data, that.data, size);
}

string :: ~string(){
    delete[] data;
}
```

OOP: Advanced class features

- Move Semantics (C++11): lvalue, rvalue

```
string a(x);                                   // Line 1
string b(x + y);                               // Line 2
string c(function_returning_a_string());      // Line 3
```

- **lvalue**: real object having an address
  - **Line 1:** `x`
- **rvalue**: temporary object – no name
  - **Line 2:** `x + y`
  - **Line 3:** `function_returning_a_string()`

OOP: Advanced class features

- Move Semantics (C++11): rvalue reference, move constructor

```
//string&& is an rvalue reference to a string
string :: string(string&& that){
    data = that.data;
    that.data = nullptr;
}
```

- **Move constructor**
  - **Shallow copy** of the argument
  - **Ownership transfer** to the new object

OOP: Advanced class features

- **Move** constructor – Stack class

```
Stack::Stack(Stack&& rhs){
    //move rhs to this
    this->mCapacity = rhs.mCapacity;
    this->mTop = rhs.mTop;
    this->mElements = rhs.mElements;

    //leave rhs in valid state
    rhs.mElements = nullptr;
    rhs.mCapacity = 0;
    rhs.mTop = 0;
}
```

OOP: Advanced class features

- Copy constructor vs. move constructor
  - Copy constructor: **deep copy**
  - Move constructor: **shallow copy + ownership transfer**

```
// constructor
string s="apple";
// copy constructor: s is an lvalue
string s1 = s;
// move constructor: right side is an rvalue
string s2 = s + s1;
```

OOP: Advanced class features

- Passing large objects

```
// C++98
// avoid expense copying

void makeBigVector(vector<int>& out){
 ...
}
vector<int> v;
makeBigVector( v );
```

```
// C++11
// move semantics

vector<int> makeBigVector(){
 …
}
auto v = makeBigVector();
```

- All STL classes have been extended to support **move semantics**
- The content of the temporary created vector is moved in v (not copied)

http://geant4.web.cern.ch/geant4/collaboration/c++11_guidelines.pdf

OOP: Advanced class features

```
class A{
     int value {10};
     static A instance;
public:
     static A& getInstance(){ return instance;}
     static A  getInstanceCopy(){ return instance;}
     int getValue() const { return value;}
     void setValue( int value ){ this->value = value;}
};
```

```
A A::instance;
int main(){
     A& v1 = A::getInstance();
     cout<<"v1: "<<v1.getValue()<<endl;
     v1.setValue(20);
     cout<<"v1: "<<v1.getValue()<<endl;
     A v2 = A::getInstanceCopy();
     cout<<"v2: "<<v2.getValue()<<endl;
     return 0;
}
```

Reference to a static variable → **lvalue**

A temporary copy of instance → **rvalue**

Output?

Modul
e 4

Object-Oriented Programming

Operator overloading

OOP: Operator overloading

Content

- Objectives
- Types of operators
- Operators
  - Arithmetic operators
  - Increment/decrement
  - Inserter/extractor operators
  - Assignment operator (copy and move)
  - Index operator
  - Relational and equality operators
  - Conversion operators

OOP: Operator overloading

Objective

- To make the class usage easier, more intuitive
  - the ability to read an object using the `extractor` operator (>>)
    - `Employee e1; cin >> e;`
  - the ability to write an object using the `inserter` operator (<<)
    - `Employee e2; cout<<e<<endl;`
  - the ability to compare objects of a given class
    - `cout<< ((e1 < e2) ? "less" : "greater");`

Operator overloading: a service to the clients of the class

OOP: Operator overloading

Limitations

- You cannot add new operator symbols. Only the existing operators can be redefined.
- Some operators cannot be overloaded:
  - `.` (member access in an object)
  - `::`(scope resolution operator)
  - **`sizeof`**
  - **`?:`**
- You cannot change the **arity** (the number of arguments) of the operator
- You cannot change the **precedence** or **associativity** of the operator

OOP: Operator overloading

How to implement?

- write a function with the name **`operator<symbol>`**
- alternatives:
  - method of your class
  - global function (usually a friend of the class)

http://en.cppreference.com/w/cpp/language/operators

OOP: Operator overloading

- There are 3 types of operators:
  - operators that must be methods (**member functions**)
    - they don't make sense outside of a class:
      - `operator=, operator(),operator[], operator->`
  - operators that must be **global functions**
    - the left-hand side of the operator is a variable of different type than your class: `operator<<, operator>>`
      - **`cout`** `<< emp;`
        - `cout: ostream`
        - `emp: Employee`
  - operators that can be **either** methods or global functions
    - **Gregoire:** “Make every operator a method unless you must make it a global function.”

OOP: Operator overloading

- Choosing argument types:
  - value vs. reference
    - Prefer passing-by-reference instead of passing-by-value.
  - `const` vs. non const
    - Prefer const unless you modify it.
- Choosing return types
  - you can specify any return type, however
    - follow the built-in types rule:
      - comparison always return `bool`
      - **arithmetic operators return an object representing the result of the arithmetic**

OOP: Operator overloading

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex{
public:
    Complex(double, double );
    void setRe( double );
    void setIm( double im);
    double getRe() const;
    double getIm() const;
    void print() const;
private:
    double re, im;
};
#endif
```

OOP: Operator overloading

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

Complex::Complex(double re, double im):re(

re),im(im) {} void Complex::setRe( double re){this-

>re = re;}

void Complex::setIm( double im){ this->im =

im;}  double Complex::getRe() const{ return

this->re;}   double Complex::getIm() const{

return this->im;}

void Complex::print()const{    cout<<re<<"+"<<im<<"i";}
```

OOP: Operator overloading

- Arithmetic operators (**member** or standalone func.)
  - unary minus
  - binary minus

```
Complex Complex::operator-() const{
    Complex temp(-this->re, -this->im);
    return temp;
}

Complex Complex::operator-( const Complex& z) const{
    Complex temp(this->re - z.re, this->im- z.im);
    return temp;
}
```

OOP: Operator overloading

- Arithmetic operators (member or **standalone** func.)
  - unary minus
  - binary minus

```
Complex operator-( const Complex& z ){
    Complex temp(-z.getRe(), -z.getIm());
    return temp;
}

Complex operator-( const Complex& z1, const Complex& z2 ){
    Complex temp(z1.getRe()-z2.getRe(), z1.getIm()-z2.getIm());
    return temp;
}
```

OOP: Operator overloading

- Increment/Decrement operators
  - postincrement:
    - `int i = 10; int j = i++; // j → 10`
  - preincrement:
    - `int i = 10; int j = ++i; // j → 11`
  - The C++ standard specifies that the prefix increment and decrement return an **lvalue** (left value).

OOP: Operator overloading

- Increment/Decrement operators (member func.)

```
Complex& Complex::operator++(){            //prefix
      (this->re)++;
      (this->im)++;
      return *this;
}

Complex   Complex::operator++( int ){ //postfix
      Complex
      temp(*this);
      (this->re)++;
      (this-
      >im)++;
      return
      temp;
}
```

Which one is more efficient? Why?

OOP: Operator overloading

- Inserter/Extractor operators (standalone func.)

```
//complex.h

class Complex {
public:
    friend ostream& operator<<(
                        ostream& os, const Complex& c);
    friend istream& operator>>(
                        istream& is, Complex& c);
    //...
};
```

OOP: Operator overloading

- Inserter/Extractor operators (standalone func.)

```
//complex.cpp

ostream& operator<<( ostream& os, const Complex& c){
      os<<c.re<<"+"<<c.im<<"i";
      return os;
}

istream& operator>>( istream& is, Complex& c){
      is>>c.re>>c.im;
      return is;
}
```

OOP: Operator overloading

- Inserter/Extractor operators
- **Syntax:**

```
ostream& operator<<( ostream& os, const T& out)
istream& operator>>( istream& is, T& in)
```

- Remarks:
  - Streams are always *passed by reference*
  - **Q:** Why should inserter operator return an **ostream&**?
  - **Q:** Why should extractor operator return an **istream&**?

OOP: Operator overloading

- Inserter/Extractor operators
- Usage:

```
Complex z1, z2;
cout<<"Read 2 complex number:";
//Extractor
cin>>z1>>z2;
//Inserter
cout<<"z1: "<<z1<<endl;
cout<<"z2: "<<z2<<endl;

cout<<"z1++: "<<(z1++)<<endl;
cout<<"++z2: "<<(++z2)<<endl;
```

OOP: Operator overloading

- **Assignment operator (=)**
  - **Q:** When should be overloaded?
  - **A:** When bitwise copy is not satisfactory (e.g. if you have dynamically allocated memory ⇒
    - when we should implement the copy constructor and the destructor too).
    - Ex. our Stack class

OOP: Operator overloading

- Assignment operator (member func.)
  - Copy assignment
  - Move assignment (since C++11)

OOP: Operator overloading

- **Copy** assignment operator (member func.)
  - **Syntax:** `X& operator=( const X& rhs);`
  - **Q:** Is the return type necessary?
    - Analyze the following example code

```
Complex z1(1,2), z2(2,3), z3(1,1);
z3 = z1;
z2 = z1 = z3;
```

OOP: Operator overloading

- **Copy** assignment operator example

```
Stack& Stack::operator=(const Stack& rhs) {
 if (this != &rhs) {
   //delete lhs - left hand side
   delete [] this->mElements;
   this->mCapacity = 0;
    this_>melements = nullptr; // in case next line throws
   //copy rhs - right hand side
   this->mCapacity = rhs.mCapacity;
   this->mElements = new double[ mCapacity ];
   int nr = rhs.mTop - rhs.mElements;
   std::copy(rhs.mElements,rhs.mElements+nr,this->mElements);
   mTop = mElements + nr;
 }
 return *this;
}
```

OOP: Operator overloading

– Copy assignment operator vs Copy constructor

```
Complex z1(1,2), z2(3,4); //Constructor
Complex z3 = z1; //Copy constructor
Complex z4(z2);   //Copy constructor
z1 = z2;          //Copy assignment operator
```

OOP: Operator overloading

- **Move** assignment operator (member func.)
  - **Syntax:** `X& operator=( X&& rhs);`
  - When it is called?

```
Complex z1(1,2), z2(3,4); //Constructor
Complex z4(z2); //Copy constructor
z1 = z2;      //Copy assignment operator
Complex z3 = z1 + z2; //Move constructor
z3 = z1 + z1;//Move assignment
```

OOP: Operator overloading

- **Move** assignment operator example

```
Stack& Stack::operator=(Stack&& rhs){
    //delete lhs - left hand side
    delete [] this->mElements;
     //move rhs to this
     this->mCapacity = rhs.mCapacity;
    this->mTop = rhs.mTop;
    this->mElements = rhs.mElements;
    //leave rhs in valid state
    rhs.mElements = nullptr;
    rhs.mCapacity = 0;
     rhs.mTop = 0;
    //return permits s1 = s2 = create_stack(4);
    return *this;
}
```

OOP: Advanced class features

- Features of a *well-behaved* C++ class (2011)
  - implicit constructor `T :: T();`
  - destructor `T :: ~T();`
  - copy constructor `T :: T( const T& );`
  - **move** constructor `T :: T( T&& );`
  - copy assignment operator
    - `T& T :: operator=( const T& );`
  - **move** assignment operator
    - `T& T :: operator=( T&& rhs );`

OOP: Operator overloading

- Subscript operator: needed for arrays (member func.)
- Suppose you want your own dynamically allocated C-style array ⇒ implement your own `CArray`

```
#ifndef CARRAY_H
#define   CARRAY_H
class CArray{
public:
    CArray( int size = 10 );
    ~CArray();
    CArray( const CArray&) = delete;
    CArray& operator=( const Carray&) = delete;
    double& operator[]( int index );
    double  operator[]( int index ) const;
private:
    double * mElems;
    int mSize;
};
#endif    /* ARRAY_H */`
```

Provides read-only access

"If the value type is known to be a built-in type, the const variant should return by value."
http://en.cppreference.com/w/cpp/language/operators.

OOP: Operator overloading

- Implementation

```
CArray::CArray( int size ){ if(
size < 0 ){
    this->size = 10;
  }
  this->mSize = size;
  this->mElems = new double[ mSize ];
}

CArray::~CArray(){
    if( mElems != nullptr ){
    delete[] mElems;  mElems =
    nullptr;
  }
}

double& CArray::operator[]( int index ){ if(
index <0 || index >= mSize ){
    throw out_of_range("");
  }
  return mElems[ index ];
```

```
double CArray::operator[](
                    int index ) const{
    if( index <0 || index >= mSize ){
        throw out_of_range("");
    }
    return mElems[ index ];
}
```

#include<stdexcept>

OOP: Operator overloading

- const vs non-const [] operator

```
void printArray(const CArray& arr, size_t size) {
    for (size_t i = 0; i < size; i++) {
        cout << arr[i] << "" ;
        // Calls the const operator[] because arr is
        // a const object.
    }
    cout << endl;
}
```

```
CArray myArray;
for (size_t i = 0; i < 10; i++) {
    myArray[i] = 100;
    // Calls the non-const operator[] because
    // myArray is a non-const object.
}
printArray(myArray, 10);
```

OOP: Operator overloading

- Relational and equality operators
  - used for **search** and **sort**
  - the container must be able to compare the stored objects

```
bool operator ==( const Point& p1, const Point& p2){
    return p1.getX() == p2.getX() && p1.getY() == p2.getY();
}


bool operator <( const Point& p1, const Point& p2){
   return p1.distance(Point(0,0)) < p2.distance(Point(0,0));
}
```

```
set<Point> p;
```

```
vector<Point> v; //...
sort(v.begin(), v.end());
```

OOP: Operator overloading

- The function call operator `()`
- Instances of classes overloading this operator behave as functions too (they are function objects = function + object)

```
#ifndef ADDVALUE_H
#define   ADDVALUE_H
class AddValue{
    int value;
public:
    AddValue( int inValue = 1);
    void operator()( int& what );
};
#endif    /* ADDVALUE_H */
```

```
#include "AddValue.h"

AddValue::AddValue( int inValue ){
    this->value = inValue;
}

void AddValue::operator()( int& what ){
    what += this->value;
}
```

OOP: Operator overloading

- The function call operator

```
AddValue func(2);
int array[]={1, 2, 3};
for( int& x : array ){
   func(x);
}
for( int x: array ){
   cout <<x<<endl;
}
```

OOP: Operator overloading

- Function call operator
  - used frequently for defining sorting criterion

```
struct EmployeeCompare{
    bool operator()( const Employee& e1, const Employee&
        e2){ if( e1.getLastName() == e2.getLastName())
            return e1.getFirstName() < e2.getFirstName();
        else
            return e1.getLastName() < e2.getLastName();

    }
};
```

OOP: Operator overloading

- Function call operator
  - sorted container

```
set<Employee, EmployeeCompare> s;

Employee e1; e1.setFirstName("Barbara");
e1.setLastName("Liskov");
Employee e2; e2.setFirstName("John");
e2.setLastName("Steinbeck");
Employee e3; e3.setFirstName("Andrew");
e3.setLastName("Foyle");
s.insert( e1 ); s.insert( e2 ); s.insert( e3 );

for( auto& emp : s){
   emp.display();
}
```

OOP: Operator overloading

- Sorting elements of a given *type*:
  - **A.** override operators: **<**, **==**
  - **B.** define a function object containing the comparison
- **Which one to use?**
  - **Q:** How many sorted criteria can be defined using method **A**?
  - **Q:** How many sorted criteria can be defined using method **B**?

OOP: Operator overloading

- Writing conversion operators

```
class Complex{
public:
   operator string() const;
   //
};

Complex::operator string() const{
    stringstream ss;
    ss<<this->re<<"+"<<this->im<<"i";
    return ss.str();
}
```

```
//usage
Complex z(1, 2);
string a = z;
cout<<a<<endl;
```

OOP: Operator overloading

- After templates
  - Overloading operator *
  - Overloading operator →

OOP: Review

- Find all possible errors or shortcommings!

```
class Array {
public:
  Array (int n) : rep_(new int [n]) { }
  Array (Array& rhs) : rep_(rhs.rep_) { }
  ~Array () { delete rep_; }
  Array& operator = (Array rhs) { rep_= rhs.rep_;   }
  int& operator [] (int n) { return &rep_[n]; }
private:
  int * rep_;
}; // Array
```

Source: http://www.cs.helsinki.fi/u/vihavain/k13/gea/exer/exer_2.html

Solution required!

– It is given the following program:

```
#include <iostream>

int main(){
    std::cout<<"Hello\n";
    return 0;
}
```

Modify the program *without modifying the main function* so that the output of the program would be:

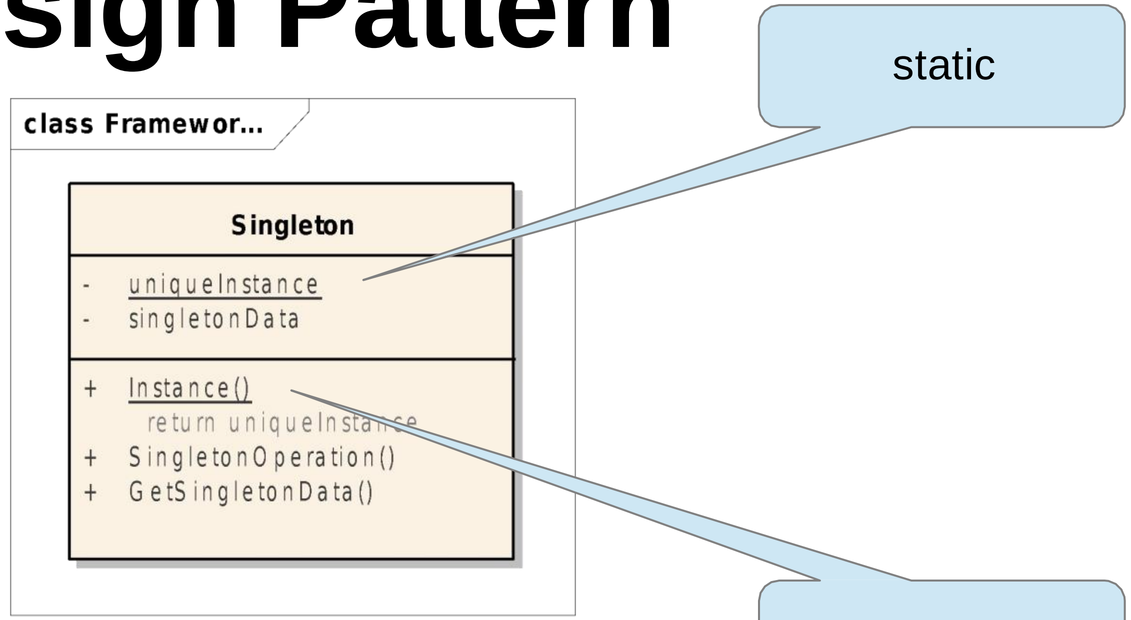
```
Start
Hello
Stop
```

Singleton Design Pattern

```
#include <string>
class Logger{
public:
    static Logger* Instance();
    bool openLogFile(std::string logFile);
    void writeToLogFile();
    bool closeLogFile();
private:
    Logger(){};  // Private so that it can  not be called
    Logger(Logger const&){}; // copy constructor is private
    Logger& operator=(Logger const&){};// assignment operator is private
    static Logger* m_pInstance;
};
```

http://www.yolinux.com/TUTORIALS/C++Singleton.html

Singleton Design Pattern



- Ensure that **only one instance** of a class is created.
- Provide a **global point of access** to the object.

Module 5

Object-Oriented Programming

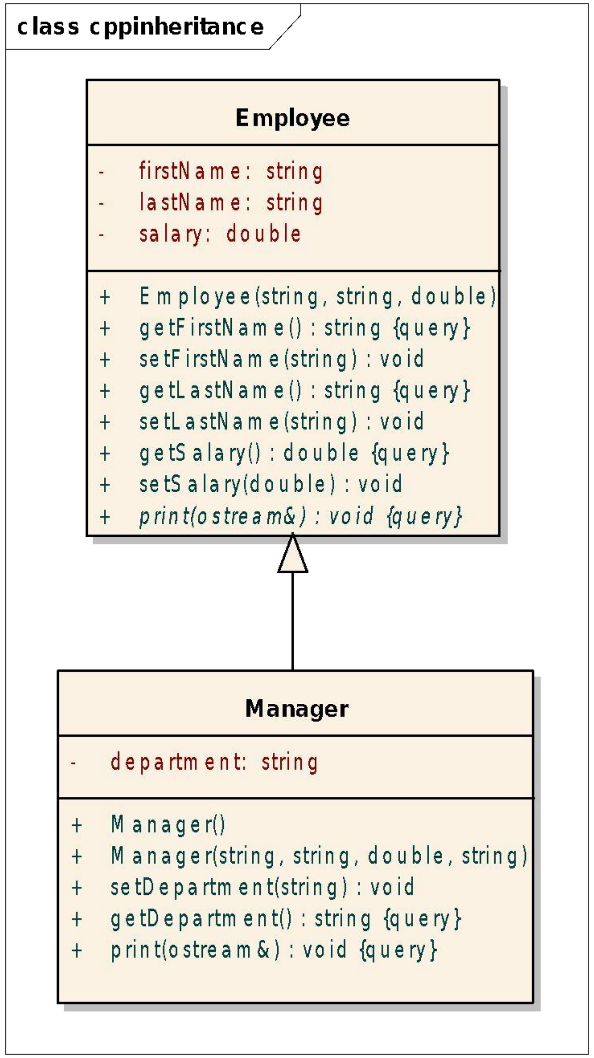
Public Inheritance

OOP: Inheritance

- Inheritance
  - *is-a* relationship - public inheritance
  - protected access
  - virtual member function
  - early (static) binding vs. late (dynamic) binding
  - abstract base classes
  - pure virtual functions
  - virtual destructor

OOP: Inheritance

- `public` inheritance
  - *is-a* relationship
  - base class: Employee
  - derived class: Manager
- You can do with inheritance
  - *add data*
    - ex. `department`
  - *add functionality*
    - ex. `getDepartment()`, `setDepartment()`
  - *modify methods' behavior*
    - ex. `print()`



OOP: Inheritance

- `protected` access
  - base class's private members can not be accessed in a derived class
  - base class's protected members can be accessed in a derived class
  - base class's public members can be accessed from anywhere

OOP: Inheritance

- `public` inheritance

```
class Employee{
public:
    Employee(string firstName = "", string lastName = "",
             double salary = 0.0) : firstName(firstName),
                                    lastName(lastName),
                                    salary(salary) {
    }
    //...
};
```

```
class Manager:public Employee{
    string department;
public:
    Manager();
    Manager( string firstName, string lastName, double salary,
             string department );
    //...
};
```

OOP: Inheritance

- Derived class's constructors

```
Manager::Manager(){
}
```

Employee's constructor invocation → Default constructor can be invoked implicitly

OOP: Inheritance

– Derived class's constructors

```
Manager::Manager(){
}
```

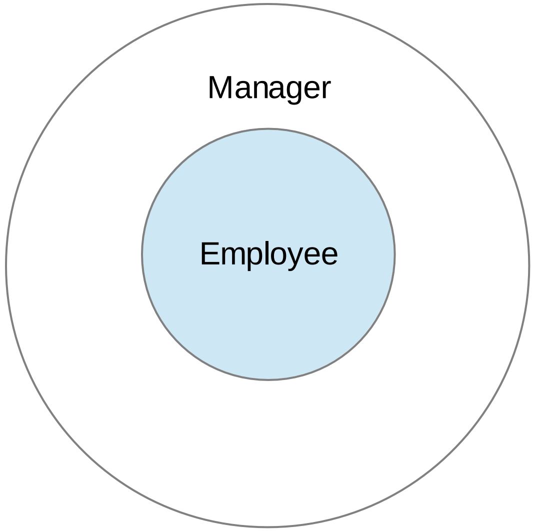
Employee's constructor invocation → Default constructor can be invoked implicitly

```
Manager::Manager(string firstName, string lastName, double salary,
                 string department):Employee(firstName, lastName, salary),
                                     artment(department){


}
```

base class's constructor invocation – *constructor initializer list*
arguments for the base class's constructor are specified in the definition of a derived class's constructor

OOP: Inheritance

- How are derived class's objects constructed?
  - *bottom up* order:
    - base class constructor invocation
    - member initialization
    - derived class's constructor block
  - destruction
    - in the opposite order



OOP: Inheritance

- Method overriding

```
class Employee{
public:
     virtual void print(ostream&) const;
};
```

```
class Manager:public Employee{
public:

    virtual void print(ostream&) const;
};
```

OOP: Inheritance

- Method overriding

```
class Employee {
public:
     virtual void print( ostream&) const;
};
```

```
void Employee::print(ostream& os ) const{
    os<<this->firstName<<" "<<this->lastName<<" "<<this->salary;
}
```

```
class Manager:public Employee{
public:
    virtual void print(ostream&) const;
};
```

```
void Manager::print(ostream& os) const{
    Employee::print(os);
    os<<" "<<department;
}
```

OOP: Inheritance

- Method overriding - virtual functions
  - non virtual functions are bound statically
    - compile time
  - virtual functions are bound dynamically
    - run time

OOP: Inheritance

- Polymorphism

```
void printAll( const vector<Employee*>& emps ){
    for( int i=0; i<emps.size(); ++i){
        emps[i]-> print(cout);
        cout<<endl;
    }
}

int main(int argc, char** argv) {
    vector<Employee*> v;
    Employee e("John", "Smith", 1000);
    v.push_back(&e);
    Manager m("Sarah", "Parker", 2000, "Sales");
    v.push_back(&m);
    cout<<endl;
    printAll( v );
    return 0;
}
```

Output:

```
John Smith 1000
Sarah Parker 2000 Sales
```

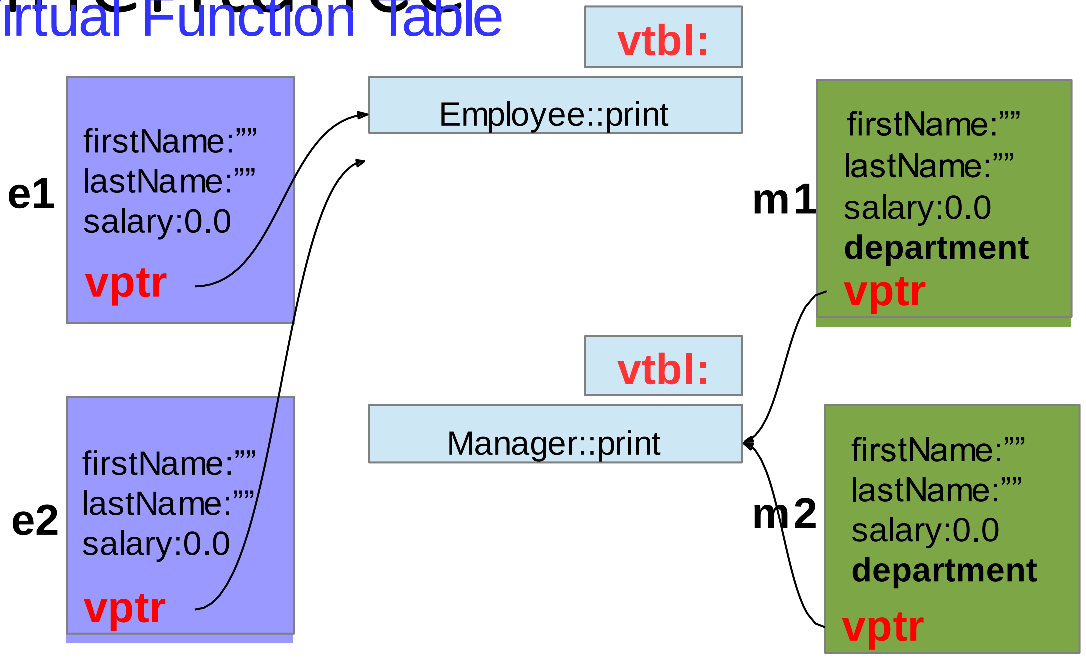
OOP: Inheritance

- Polymorphism
  - a type with virtual functions is called a polymorphic type
  - polymorphic behavior **preconditions**:
    - the member function must be virtual
    - objects must be manipulated through
      - pointers or
      - references
    - `Employee :: print( os )` static binding – no polymorphism

OOP: Inheritance

– Polymorphism – Virtual Function Table

```
class Employee{
public:
 virtual void print(ostream&) const;
     //...
};

class Manager:public Employee{
 virtual void print(ostream&) const;
     //...
};
Employee e1, e2;
Manager m1, m2;
```



Discussion!!!

```
Employee * pe;
pe = &e1; pe->print(); //???
pe = &m2; pe->print(); //???
```

Each class with virtual functions has its own virtual function table (vtbl).

RTTI – Run-Time Type Information

dynamic_cast<>(pointer)

```
class Base{};
class Derived : public Base{};

Base* basePointer = new Derived();
Derived* derivedPointer = nullptr;

//To find whether basePointer is pointing to Derived type of object

derivedPointer =
dynamic_cast<Derived*>(basePointer); if
(derivedPointer != nullptr){
   cout << "basePointer is pointing to a Derived class object";
}else{
   cout << "basePointer is NOT pointing to a Derived class object";
}
```

Java:
instanceof

RTTI – Run-Time Type Information

dynamic_cast<>(reference)

```
class Base{};
class Derived : public Base{};

Derived derived;
Base& baseRef = derived;

// If the operand of a dynamic_cast to a reference isn't of the expected type,
//  a bad_cast exception is thrown.

try{
      Derived& derivedRef = dynamic_cast<Derived&>(baseRef);
} catch( bad_cast ){
      // ..
}
```

OOP: Inheritance

- Abstract classes
  - used for representing abstract concepts
  - used as base class for other classes
  - no instances can be created

OOP: Inheritance

- Abstract classes – pure virtual functions

```
class Shape{ // abstract class
    public:
        virtual void rotate(int)  = 0;      // pure virtual function
        virtual void draw()  = 0;  // pure virtual function
        // ...
};
```

```
Shape s; //???
```

OOP: Inheritance

– Abstract classes – pure virtual functions

```
class Shape{ // abstract class
    public:
        virtual void rotate(int)  = 0;      // pure virtual function
        virtual void draw()  = 0;  // pure virtual function
        // ...
};
```

```
Shape s; //Compiler error
```

OOP: Inheritance

- Abstract class → concrete class

```
class Point{ /* ... */ };
class Circle : public Shape {
    public:
        void rotate(int);          // override Shape::rotate
        void draw();               // override Shape::draw
        Circle(Point p, int r) ;
    private:
        Point center;
        int radius;
};
```

OOP: Inheritance

- Abstract class → abstract class

```
class Polygon : public Shape{
public:
     // draw() and rotate() are not overridden

};
```

```
Polygon p; //Compiler error
```

OOP: Inheritance

- Virtual destructor
  - Every class having at least one virtual function should have virtual destructor. Why?

```
class X{
public:
    // ...
    virtual ~X();
};
```

OOP: Inheritance

- Virtual destructor

```
void deleteAll( Employee ** emps, int size){
    for( int i=0; i<size; ++i){
        delete emps[ i ];
    }
    delete [] emps;
}

 // main
 Employee ** t = new Employee *[ 10 ];
 for(int i=0; i<10; ++i){
    if( i % 2 == 0 )
        t[ i ] = new Employee();
    else
        t[ i ] = new Manager();
 }
 deleteAll( t, 10);
```

Which destructor is invoked?

Module 6

Object-Oriented Programming

Object relationships

OOP: Object relationships

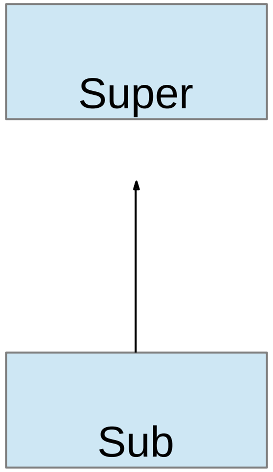
- The ***is-a*** relationship
  - Private inheritance
  - Multiple inheritance

- The ***has-a*** relationship
  - Association
    - Composition (strong containment)
    - Aggregation (weak containment)

OOP: Object relationships

- The *is-a* relationship – *Client's view (1)*
  - works in only *one direction:*
    - every **Sub** object **is** also **a Super** one
    - but **Super** object **is not a Sub**

```
void foo1( const Super& s );
void foo2( const Sub& s);
Super super;
Sub sub;

foo1(super); //OK
foo1(sub);    //OK
foo2(super); //NOT OK
foo2(sub);    //OK
```

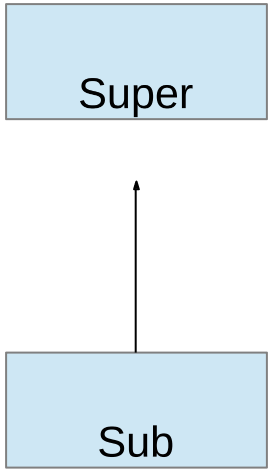


OOP: Object relationships

- The *is-a* relationship – *Client's view (2)*

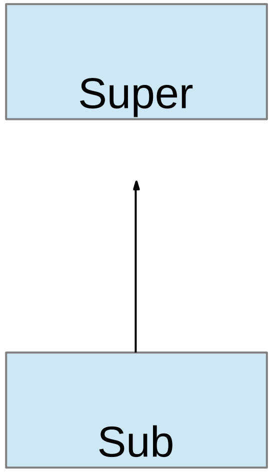
```
class Super{
public:
   virtual void method1();
};
class Sub : public Super{
public:
   virtual void method2();
};
```

```
Super * p= new Super();
p->method1(); //OK

p = new Sub();
p->method1(); //OK
p->method2(); //NOT OK
((Sub *)p)->method2();//OK
```



OOP: Object relationships

- The *is-a* relationship – *Sub-class's view*



  - the `Sub` class augments the `Super` class by **adding additional methods**
  - the `Sub` class **may override** the `Super` class **methods**
  - the subclass can use all the public and protected members of a superclass.

OOP: Object relationships

- The *is-a* relationship: *preventing inheritance* **C++*11***
  - `final` classes – cannot be extended

```
class Super final
{

};
```

OOP: Object relationships

- The *is-a* relationship: *a client's view of overridden methods(1)*
  - *polymorphism*

```
class Super{
public:
    virtual void method1();
};
class Sub : public Super{
public:
    virtual void method1();
};
```

```
Super super;
super.method1(); //Super::method1()

Sub sub;
sub.method1();   //Sub::method1()

Super& ref =super;
ref.method1();   //Super::method1();

ref = sub;
ref.method1();   //Sub::method1();

Super* ptr =&super;
ptr->method1(); //Super::method1();

ptr = &sub;
ptr->method1();   //Sub::method1();
```

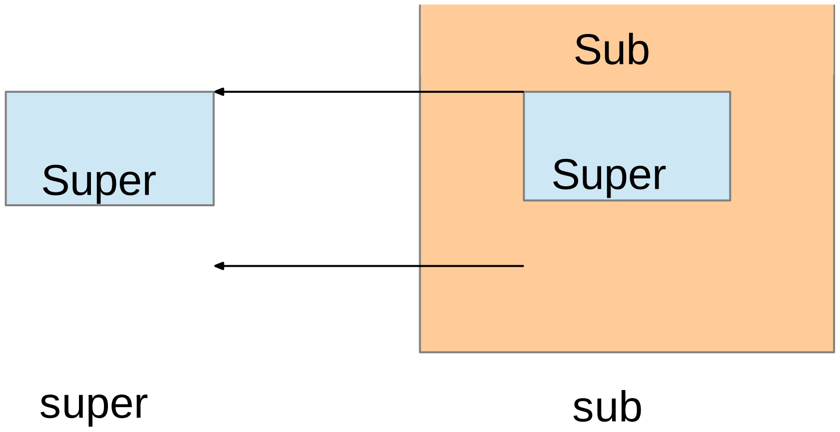
OOP: Object relationships

– The *is-a* relationship: *a client's view of overridden methods(2)*

- object slicing

```
class Super{
public:
    virtual void method1();
};
class Sub : public Super{
public:
    virtual void method1();
};
```

```
Sub sub;
Super super = sub;
super.method1(); // Super::method1();
```



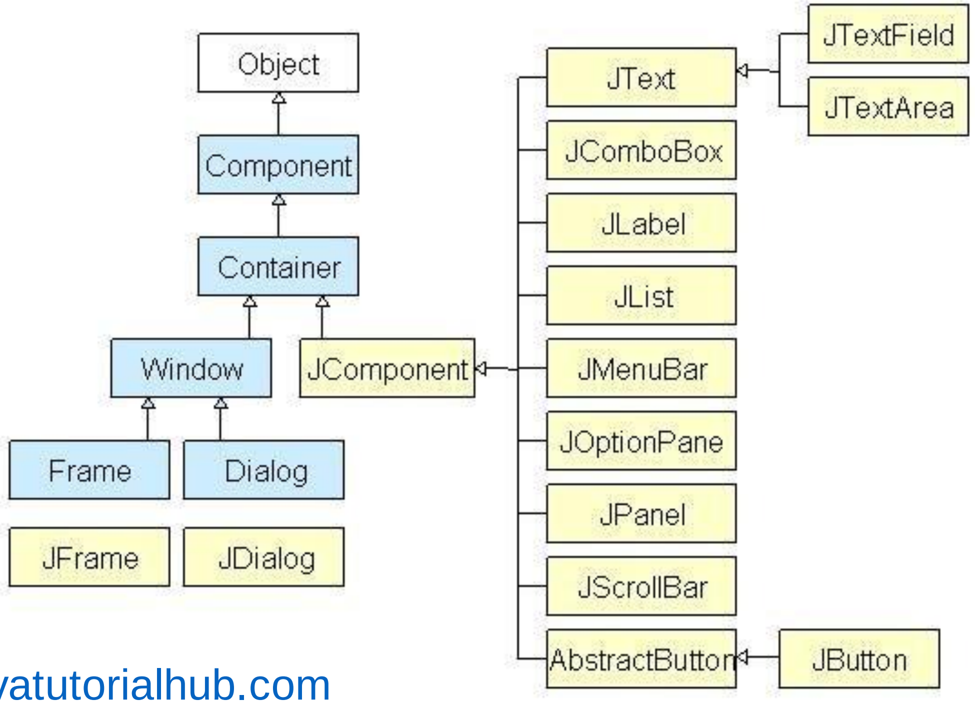
OOP: Object relationships

- The *is-a* relationship: *preventing method overriding* ***C++11***

```
class Super{
public:
    virtual void method1() final;
};
class Sub : public Super{
public:
    virtual void method1(); //ERROR
};
```

OOP: Object relationships

- Inheritance for polymorphism



www.javatutorialhub.com

OOP: Object relationships

- The *has-a* relationship

OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**

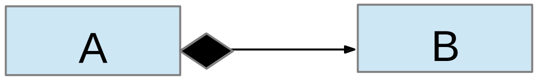
```
class B;

class
A{
private
:
    B b;
};
```

```
class B;

class
A{
private
:
    B*  b;
};
```

```
class B;

class
A{
private
:
    B&  b;
};
```

OOP: Object relationships

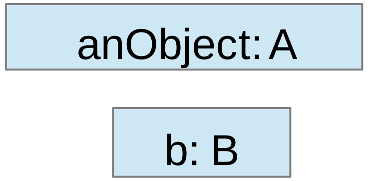
- Implementing the *has-a* relationship



  - An object **A** has an object **B**
    - **strong containment (composition)**

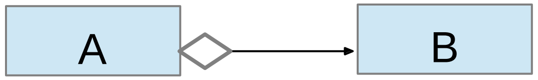
```
class B;

class A{
private:
    B b;
};
```

```
A anObject;
```



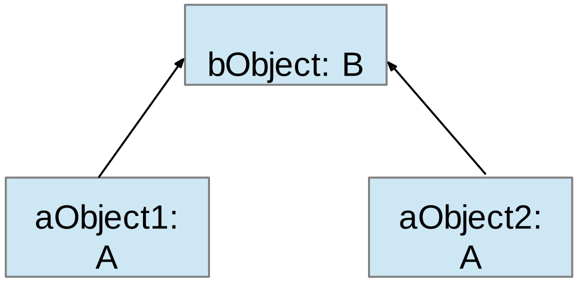
OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**
    - **weak containment (aggregation)**




```
class B;

class A{
private:
    B& b;
public:
    A( const B& pb):b(pb){}
};
```

```
B bObject;
A aObject1(bObject);
A aObject2(bObject);
```



OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**

weak containment

```
class B;

class A{
private:
     B* b;
public:
     A( B* pb):b( pb ){}
};
```

strong containment

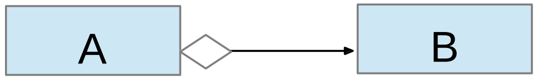
```
class B;

class A{
private:
     B* b;
public:
     A(){
          b = new B();
     }
     ~A(){
          delete b;
     }
};
```

OOP: Object relationships

– Implementing the *has-a* relationship

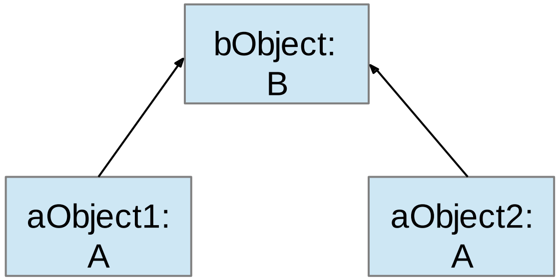
- An object **A** has an object **B**

weak containment



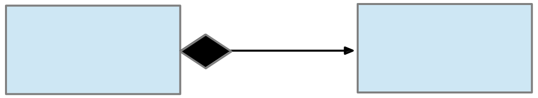

```
class B;

class A{
private:
     B* b;
public:
     A( B* pb):b( pb ){}
};
```

```
Usage:
     B bObject;
     A aObject1(&bObject);
     A aObject2(&bObject);
```



OOP: Object relationships

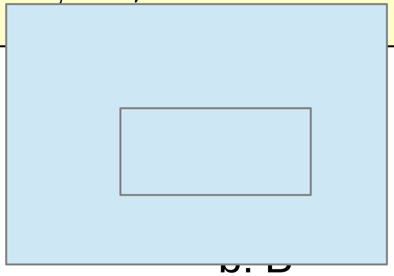
– Implementing the *has-a* relationship



- An object **A** has an object **B strong containment**

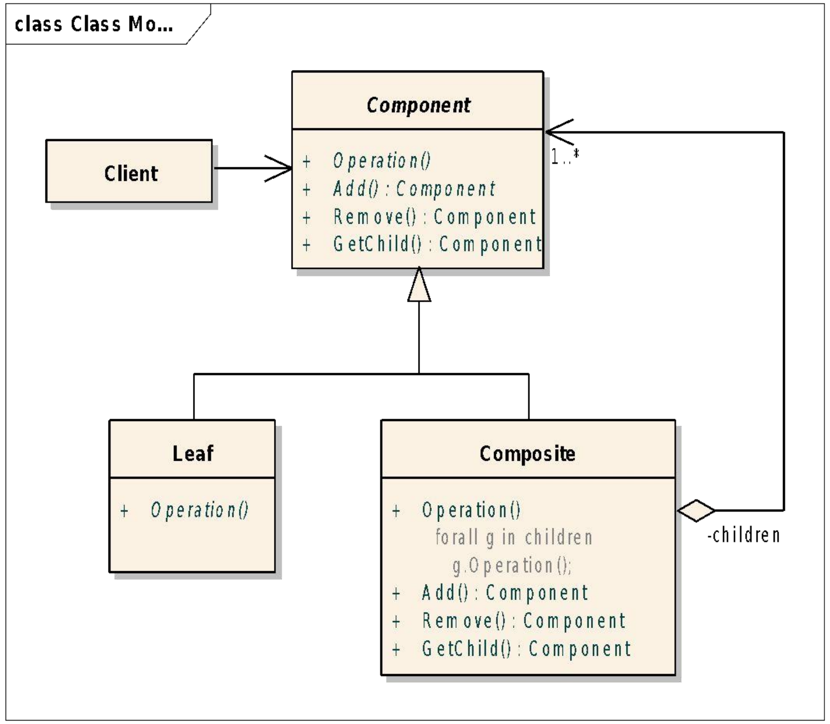
```
class B;

class
A{
priva
te:
      B*
b;
public
:
      A(){
            b = new B();
      }
      ~A(){
            delete b;
```

```
Usage:
      A aObject;
```

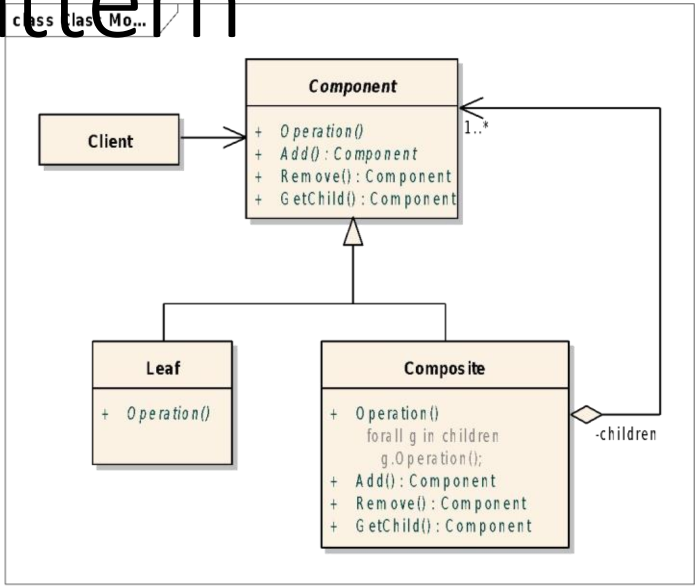


OOP: Object relationships

- Combining the *is-a* and the *has-a* relationships



Composite Design Pattern



- Compose objects into tree structures to represent **part-whole hierarchies.**
- Lets clients treat **individual objects** and **composition of objects uniformly.**

Composite Design Pattern

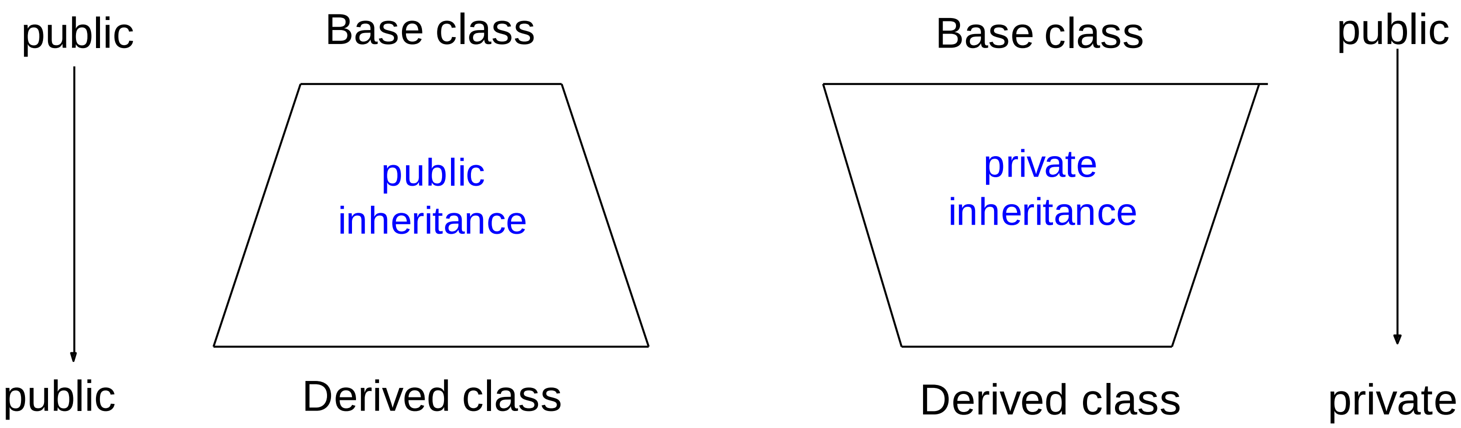
Examples:

- Menu – MenuItem: Menus that contain menu items, each of which could be a menu.
- Container – Element: Containers that contain Elements, each of which could be a Container.
- GUI Container – GUI component: GUI containers that contain GUI components, each of which could be a container

Source: http://www.oodesign.com/composite-pattern.html

Private Inheritance

– another possibility for *has-a* relationship



Derived class **inherits** the base class behavior

Derived class **hides** the base class behavior

Private Inheritance

```
template <typename T>
class MyStack : private vector<T> {
public:
    void push(T elem) {
        this->push_back(elem);
    }
    bool isEmpty() {
        return this->empty();
    }
    void pop() {
        if (!this->empty())this->pop_back();
    }
    T top() {
        if (this->empty()) throw out_of_range("Stack is empty");
        else return this->back();
    }
};
```

Why is **public inheritance** in this case dangerous???

Non-public Inheritance

- it is very rare;
- use it cautiously;
- most programmers are not familiar with it;

What does it print?

```
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
            cout<<"Super"<<endl;
    }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d){
            cout<<"Sub"<<endl;
    }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

What does it print?

```
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
            cout<<"Super"<<endl;
     }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d){
            cout<<"Sub"<<endl;
     }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

creates a new method, instead of overriding the method

The `override` keyword C++11

```
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
            cout<<"Super"<<endl;
     }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d) const override{
            cout<<"Sub"<<endl;
     }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

Module 2

Object-Oriented Programming

Classes and Objects

Object-Oriented Programming (OOP)

Content

- Classes and Objects
- Advanced Class Features
- Operator overloading
- Object Relationships
- Abstraction

OOP: Classes and Objects

Content

- Members of the class. Access levels. Encapsulation.
- Class: **interface + implementation**
- Constructors and **destructors**
- **`const`** member functions
- Constructor initializer
- Copy constructor
- Object's lifecycle

OOP: Types of Classes

Types of classes:

- **Polymorphic** Classes – *designed for extension*
  - `Shape`, `exception`,...
- **Value** Classes – *designed for storing values*
  - `int, complex<double>, ...`
- **RAII** (**R**esource **A**cquisition **I**s **I**nitialization) Classes –
- (encapsulate a **resource** into a class → resource lifetime object lifetime)
  - `thread, unique_ptr, ...`

What type of resource?

OOP: Classes and objects

Class = Type (Data + Operations)

- Members of the class
- **Data:**
  - data members (properties, attributes)
- **Operations:**
  - methods (behaviors)
- Each member is associated with an **access level**:
  - `private` **-**
  - `public` **+**
  - `protected` **#**

OOP: Classes and objects

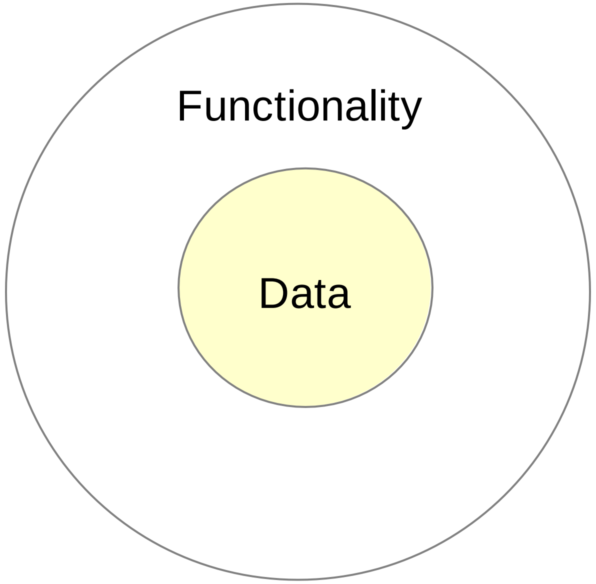
Object = Instance of a class

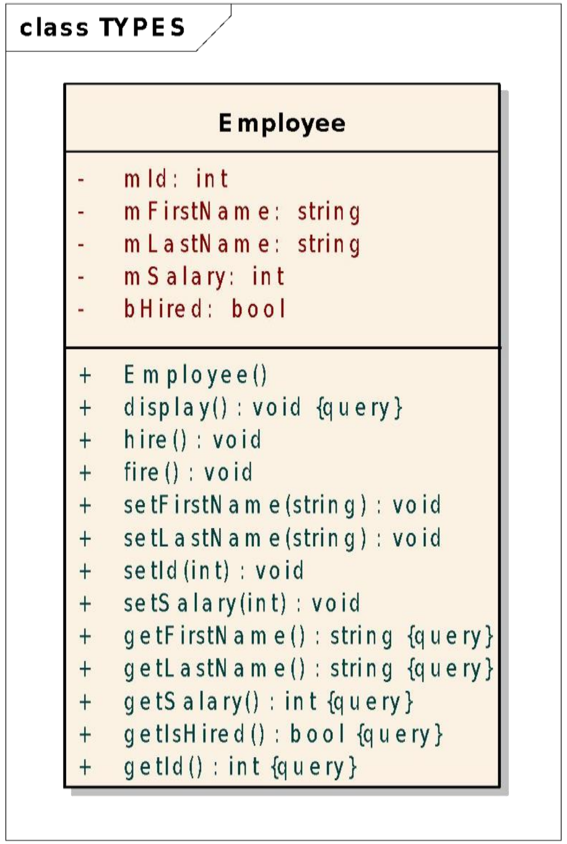
- An employee object: `Employee emp;`
  - **Properties** are the characteristics that describe an object.
    - *What makes this object different?*
      - `id, firstName, lastName, salary, hired`
  - **Behaviors** answer the question:
    - *What can we do to this object?*
      - `hire(), fire(), display(), get` and `set` data members

OOP: Classes and objects

Encapsulation

- an object encapsulates *data* and *functionality*.





OOP: Classes and objects

Class creation

- class **declaration** - *interface*
  - `Employee.h`
- class **definition** – *implementation*
  - `Employee.cpp`

OOP: Classes and objects

Employee.h

```
class Employee{
public:
    Employee();
    void display() const;
    void hire();
    void fire();
    // Getters and setters
    void setFirstName( string inFirstName );
    void setLastName ( string inLastName );
    void setId( int inId );
    void setSalary( int inSalary );
    string getFirstName() const;
    string getLastName() const;
    int getSalary() const;
    bool getIsHired() const;
     int getId() const;
private:
    int mId;
    string mFirstName;
    string mLastName;
    int mSalary;
    bool bHired;
};
```

Methods' declaration

Data members

OOP: Classes and objects

The Constructor and the object's state

- The **state of an object** is defined by its data members.
- The **constructor** is responsible for the **initial state** of the object

```
Employee :: Employee() : mId(-1),
                          mFirstName(""),
                          mLastName(""),
                          mSalary(0),
                          bHired(false){
}
```

Members are initialized through the constructor initializer list

```
Employee :: Employee() {
        mId = -1;
        mFirstName="";
        mLastName="";
        mSalary =0;
        bHired = false;
}
```

Members are assigned

Only constructors can use this initializer-list syntax!!!

OOP: Classes and objects

Constructors

- *responsibility:* data members initialization of a class object
- invoked automatically for each object
- have the *same name* as the class
- have *no return type*
- a class can have *multiple constructors* (function **overloading**)
- may not be declared as `const`
  - constructors can write to const objects

OOP: Classes and objects

Member initialization (C++11)

```
class C{
   string s ("abc");
   double d = 0;
   char * p {nullptr};
   int y[4] {1,2,3,4};
public:
   C(){}
};
```

```
class C{
   string s;
   double d;
   char * p;
   int y[5];
public:
   C():s("abc"),
   d(0.0),p(nullptr),
   y{1,2,3,4} {}
};
```

Compiler

OOP: Classes and objects

Defining a member function

- Employee.cpp
- A `const` **member function** cannot change the object's state, can be invoked on const objects

```
void Employee::hire(){
    bHired = true;
}

string Employee::getFirstName() const{
    return mFirstName;
}
```

OOP: Classes and objects

Defining a member function

```
void Employee::display() const {
    cou  << "Employee: " << getLastName() << ",
    t    << "
         << getFirstName() << endl;
    cou     "-------------------------" << endl;
    t
    cout << (bHired ? "Current Employee" :
                      "Former Employee") <<
                      endl;
    cout << "Employee ID: " << getId() << endl;
    cout << "Salary: " << getSalary() << endl;
    cout << endl;
}
```

OOP: Classes and objects

TestEmployee.cpp

- Using `const` member functions

```
void foo( const Employee& e){
   e.display();  // OK. display() is a const member function
   e.fire();     // ERROR. fire() is not a const member function
}

int main() {
    Employee emp;
    emp.setFirstName("Robert");
    emp.setLastName("Black");
    emp.setId(1);
    emp.setSalary(1000);
    emp.hire();
    emp.display();
    foo( emp );
    return 0;
}
```

OOP: Classes and objects

Interface: **Employee.h**

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

class Employee{
public:
    Employee();
    //...
protected:
    int mId;
    string mFirstName;
    string mLastName;
    int mSalary;
    bool bHired;
};

#endif
```

Implementation: **Employee.cpp**

```
#include "Employee.h"


Employee::Employee() :
  mId(-1),
  mFirstName(""),
  mLastName(""),
  mSalary(0),
  bHired(false){
}

string Employee::getFirstName() const{
    return mFirstName;
}
/
/ ...
```

OOP: Classes and objects

Object life cycles:

- creation
- assignment
- destruction

OOP: Classes and objects

Object creation:

```
int main() {
    Employee emp;
    emp.display();

    Employee *demp = new Employee();
    demp->display();
    // ..
    delete demp;
    return 0;
}
```

object's lifecycle

- all its *embedded objects* are also created

OOP: Classes and objects

Object creation – constructors:

- *default constructor* (0-argument constructor)

```
Employee :: Employee() : mId(-1), mFirstName(""),
mLastName(""), mSalary(0), bHired(false){
}
```

```
Employee :: Employee() {
}
```

```
• Employee employees[ 10 ];
• vector<Employee>  emps(10);
```

- memory allocation
- constructor call on each allocated object

OOP: Classes and objects

Object creation – constructors:

- *Compiler-generated default constructor*

- if a class *does not specify* any constructors, the *compiler will generate* one that does not take any arguments

```
class Value{
public:
    void setValue( double inValue);
    double getValue() const;
private:
    double value;
};
```

OOP: Classes and objects

Constructors: `default` and `delete` specifiers (C++ 11)

```
class X{
    int i = 4;
    int j {5};
public:
    X(int a) : i{a} {} // i = a, j = 5
    X() = default;       // i = 4, j = 5
};
```

Explicitly forcing the automatic generation of a **default** constructor by the compiler.

OOP: Classes and objects

Constructors: `default` and `delete` specifiers (C++ 11)

```
class X{
public:
    X( double ){}
};

X x2(3.14); //OK
X x1(10); //OK
```

`int` → `double` conversion

```
class X{
public:
    X( int )= delete;
    X( double );
};
X x1(10); //ERROR
X x2(3.14); //OK
```

OOP: Classes and objects

Best practice: *always provide default values for members!* C++ 11

```
struct
Point{
int x, y;
    Point ( int x = 0, int y = 0 ): x(x), y(y){}
};
class Foo{
    int
    i

    {};
    double
    d {};
    char c
    {};
    Point p {};
public:
    void print(){
        cout        <<"i:
        "<<i<<endl;   cout
        <<"d:  "<<d<<endl;
        cout        <<"c:
        "<<c<<endl;
        cout <<"p: "<<p.x<<", "<<p.y<<endl;
```

```
int main() {
    Foo f;
    f.print(
    );
    return
    0;
}
```

```
p: 0, 0
```

OOP: Classes and objects

Constructor initializer

```
class ConstRef{
public:
    ConstRef( int& );
private:
    int mI;
    const int mCi;
    int& mRi;
};

ConstRef::ConstRef( int& inI ){
    mI = inI;  //OK
    mCi = inI; //ERROR: cannot assign to a const
    mRi = inI; //ERROR: uninitialized reference member
}
```

```
ConstRef::ConstRef( int& inI ): mI( inI ), mCi( inI ), mRi( inI ){}
```

ctor initializer

OOP: Classes and objects

Constructor initializer

- data types that must be initialized in a ctor-initializer
  - `const` data members
  - reference data members
  - object data members having no default constructor
  - superclasses without default constructor

OOP: Classes and objects

A *non-default* Constructor

```
Employee :: Employee( int inId, string inFirstName,
                      string inLastName,
                      int inSalary, int inHired) :
          mId(inId), mFirstName(inFirstName),
          mLastName(inLastName), mSalary(inSalary),
          bHired(inHired){
}
```

OOP: Classes and objects

Delegating Constructor (C++11)

```
class SomeType{
  int number;

public:
  SomeType(int newNumber) : number(newNumber) {}
  SomeType() : SomeType(42) {}
};
```

OOP: Classes and objects

Copy Constructor

`Employee emp1(1, "Robert", "Black", 4000, true);`

- called in one of the following cases:
  - `Employee emp2( emp1 ); //copy-constructor called`
  - `Employee emp3 = emp2;  //copy-constructor called`
  - `void foo( Employee emp );//copy-constructor called`
- if you don't define a copy-constructor explicitly, the compiler creates one for you
  - this performs a **bitwise** copy

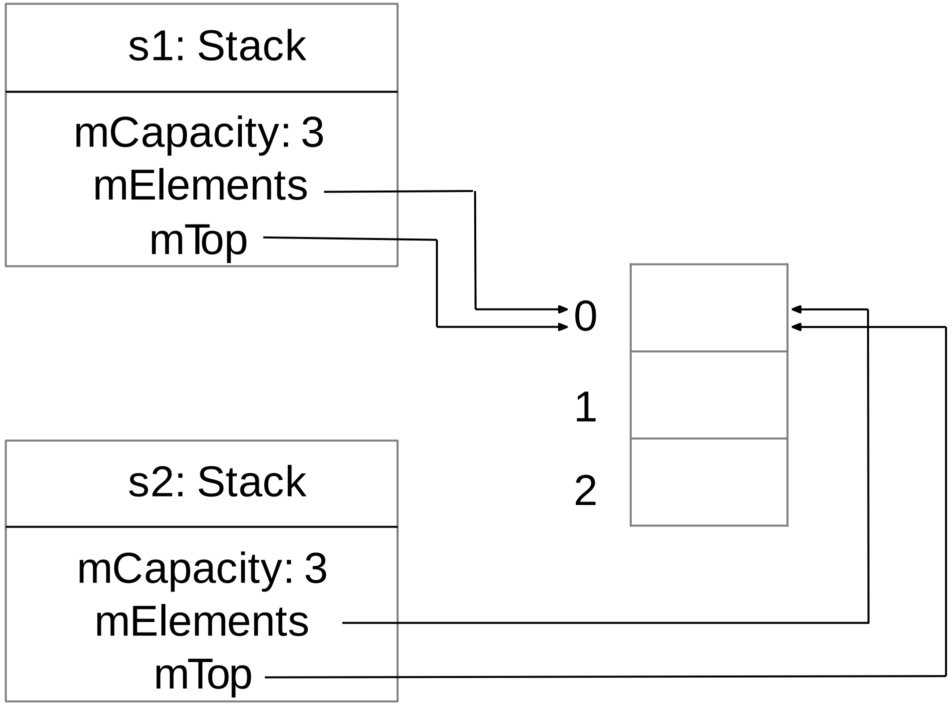
OOP: Classes and objects

```
//Stack.h

#ifndef STACK_H
#define  STACK_H

class Stack{
public:
    Stack( int inCapacity );
    void push( double inDouble );
    double top() const;
    void pop();
    bool isFull() const;
    bool isEmpty()const;

private:
    int mCapacity;
    double * mElements;
    double * mTop;
};
#endif   /* STACK_H */
```

```
//Stack.cpp

#include  "Stack.h"

 Stack::Stack( int inCapacity ){
     mCapacity = inCapacity;
     mElements = new double [ mCapacity ];
     mTop = mElements;
 }

 void Stack::push( double inDouble ){
     if( !isFull()){
         *mTop = inDouble;
         mTop++;
     }
 }
```

OOP: Classes and objects

```
//TestStack.cpp
#include "Stack.h"

int main(){
     Stack s1(3);
    Stack s2 = s1;
    s1.push(1);
    s2.push(2);

    cout<<"s1: "<<s1.top()<<endl;
    cout<<"s2: "<<s2.top()<<endl;
}
```



OOP: Classes and objects

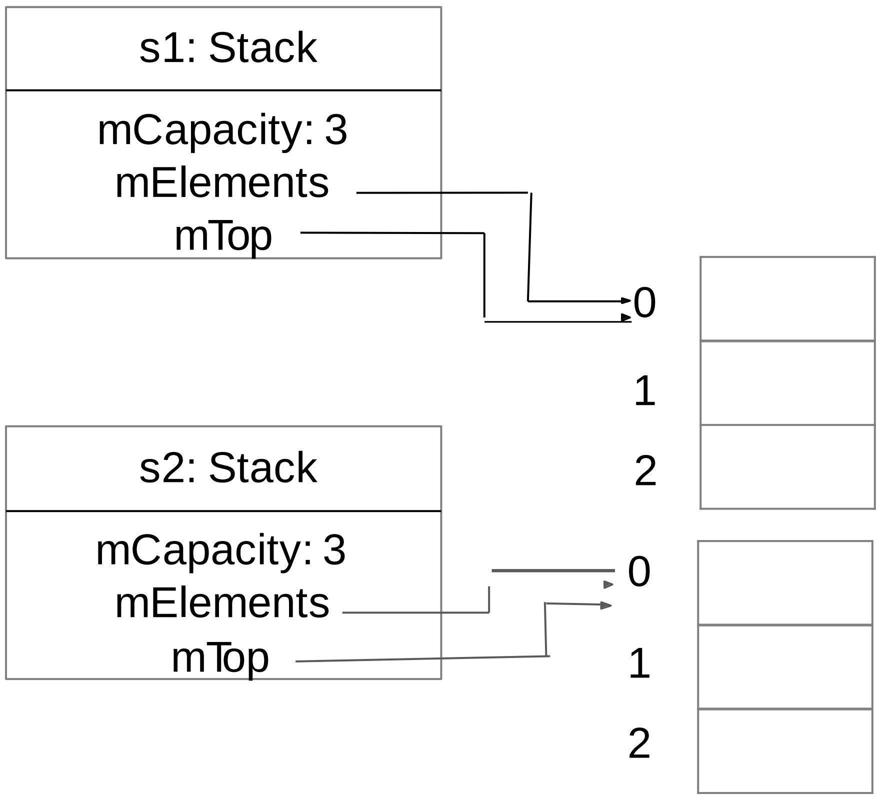
Copy constructor: **T (const T&)**

```
//Stack.h

#ifndef STACK_H
#define  STACK_H

class Stack{
public:
    //Copy constructor
    Stack( const Stack& );
private:
    int mCapacity;
    double * mElements;
    double * mTop;
};
#endif   /* STACK_H */
```

```
//Stack.cpp

#include "Stack.h"

Stack::Stack( const Stack& s ){
     mCapacity = s.mCapacity;
     mElements = new double[ mCapacity ];
     int nr = s.mTop - s.mElements;
     for( int i=0; i<nr; ++i ){
        mElements[ i ] = s.mElements[ i ];
     }
     mTop = mElements + nr;
 }
```

OOP: Classes and objects

```
//TestStack.cpp
#include "Stack.h"

int main(){
     Stack s1(3);
    Stack s2 = s1;
    s1.push(1);
    s2.push(2);

    cout<<"s1: "<<s1.top()<<endl;
    cout<<"s2: "<<s2.top()<<endl;
}
```



OOP: Classes and objects

Destructor

- when an object is destroyed:
  - the object's destructor is automatically invoked,
  - the memory used by the object is freed.
- each class has one destructor
- usually place to perform cleanup work for the object
- if you don't declare a destructor → the compiler will generate one, which destroys the object's member

OOP: Classes and objects

Destructor

- Syntax: `T :: ~T();`

```
Stack::~Stack(){
    if( mElements != nullptr ){
        delete[] mElements;
        mElements = nullptr;
    }
}
```

```
{    // block begin
     Stack s(10);          // s: constructor
    Stack* s1 = new Stack(5);// s1: constructor
    s.push(3);
     s1->push(10);
     delete s1;              //s1: destructor
     s.push(16);
}   // block end            //s: destructor
```

OOP: Classes and objects

Default parameters

- if the user specifies the arguments → the defaults are ignored
- if the user omits the arguments → the defaults are used
- the default parameters are specified **only** in the ***method declaration*** (not in the definition)

```
//Stack.h
class Stack{
public:
    Stack( int inCapacity = 5 );
     ..
};
//Stack.cpp
Stack::Stack( int inCapacity ){
     mCapacity = inCapacity;
     mElements = new double [ mCapacity ];
     mTop = mElements;
}
```

```
//TestStack.cpp

Stack   s1(3);   //capacity:  3
Stack   s2;       //capacity:  5
Stack s3( 10 ); //capacity: 10
```

OOP: Classes and objects

The `this` pointer

- every method call passes a pointer to the object for which it is called as *hidden parameter* having the name `this`
- Usage:
  - for disambiguation

```
Stack::Stack( int mCapacity ){
      this → mCapacity = mCapacity;
      //..
}
```

OOP: Classes and objects

Programming task [Prata]

```
class Queue
{
   enum {Q_SIZE = 10};
private:
   // private representation to be developed later
public:
   Queue(int qs = Q_SIZE); // create queue with a qs limit
   ~Queue();
   bool isempty() const;
   bool isfull()  const;
   int queuecount() const;
   bool enqueue(const Item &item); // add item to end
   bool dequeue(Item &item); // remove item from front
};
```

OOP: Classes and objects

Programming task [Prata]

```
class Queue
{
private:
    // class scope definitions

    // Node is a nested structure definition local to this class
    struct Node { Item item; struct Node * next;};
    enum {Q_SIZE = 10};

    // private class members
    Node * front; // pointer to front of Queue
    Node * rear; // pointer to rear of Queue
    int items; // current number of items in Queue
    const int qsize; // maximum number of items in Queue

};
```

Module 3

Object-Oriented Programming

Advanced Class Features

OOP: Advanced class features

Content

- Inline functions
- Stack vs. Heap
- Array of objects vs. array of pointers
- Passing function arguments
- Static members
- Friend functions, friend classes
- Nested classes
- Move semantics (C++11)

OOP: Advanced class features

`Inline` functions

- designed to speed up programs (like macros)
- the compiler replaces the function call with the function code (no function call!)
- advantage: *speed*
- disadvantage: *code bloat*
  - ex. 10 function calls → 10 * function's size

OOP: Advanced class features

How to make a function `inline`?

- use the `inline` keyword either in function declaration or in function definition
- both member and standalone functions can be `inline`
- common practice:
  - place the implementation of the `inline` function into the header file
- only small functions are eligible as `inline`
- the compiler may completely ignore your request

OOP: Advanced class features

`inline` function examples

```
inline double square(double a){
   return a * a;
}

class Value{
   int value;
public:
   inline int getValue()const{ return value; }

    inline void setValue( int value ){
       this->value = value;
   }
};
```

OOP: Advanced class features

- Stack vs. Heap
- Heap – Dynamic allocation

```
void draw(){
    Point * p = new Point();
    p->move(3,3);
    //...
    delete p;
}
```

- Stack – Automatic allocation

```
void draw(){
    Point p;
    p.move(6,6);
    //...
}
```

OOP: Advanced class features

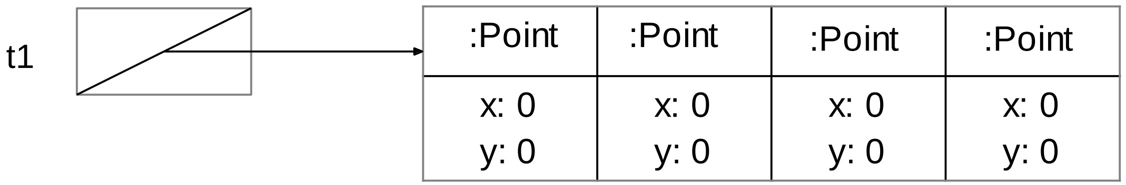
Array of objects

```
class Point{
     int x, y;
public:
     Point( int x=0, int y=0);
     //...
};
```

What is the difference between these two arrays?

Point * t1 = new Point[4];

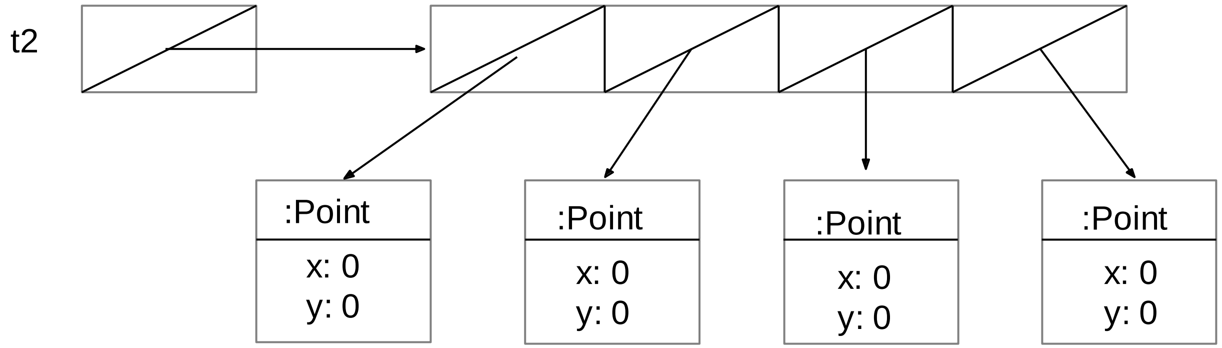
Point t1[4];



OOP: Advanced class features

Array of pointers

```
Point ** t2 = new Point*[ 4 ];
for(int i=0; i<4; ++i ){
     t2[i] = new Point(0,0);
}
for( int i=0; i<4; ++i ){
     cout<<*t2[ i ]<<endl;
}
```



OOP: Advanced class features

Static members:

- `static` methods
- `static` data

- Functions belonging to a *class scope* which don't access object's data can be `static`
- Static methods can't be `const` methods (they do not access object's state)
- They are not called on specific objects ⇒ they have no `this` pointer

OOP: Advanced class features

- Static members

```
//Complex.h

class Complex{
public:
    Complex(int re=0, int im=0);
    static int getNumComplex();
    // ...
private:
   static int num_complex;
   double re, im;
};
```

instance counter

initializing static class member

```
//Complex.cpp

int Complex::num_complex = 0;

int Complex::getNumComplex(){
    return num_complex;
}

Complex::Complex(int re, int im){
    this->re = re;
    this->im = im;
    ++num_complex;
}
```

OOP: Advanced class features

- Static method invocation

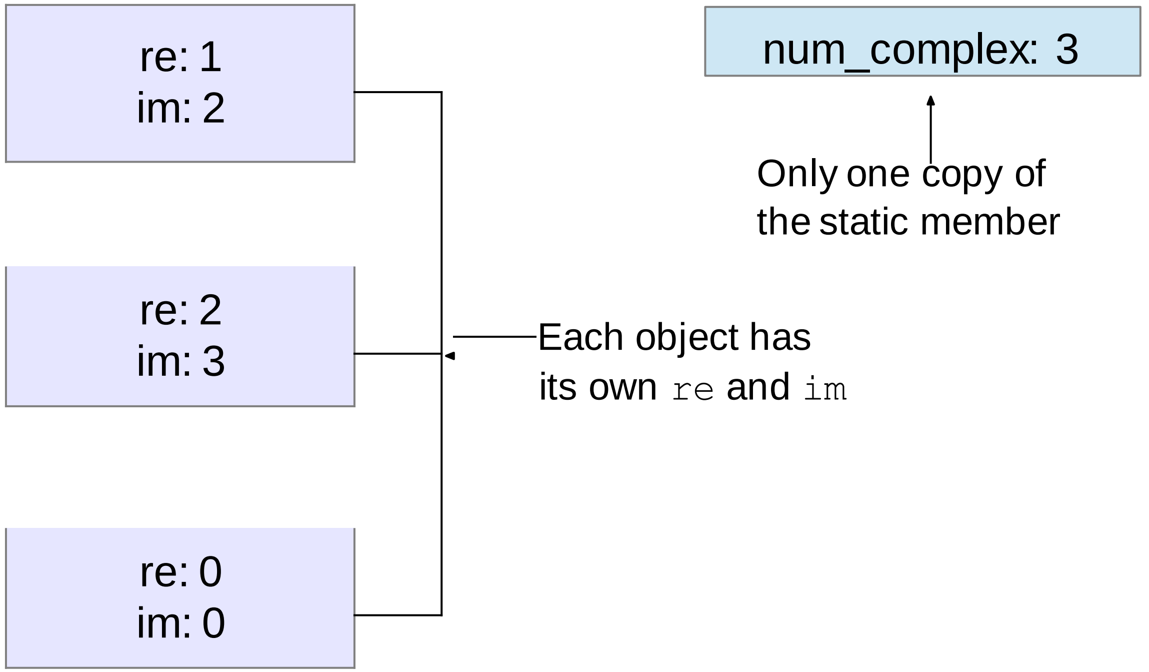
elegant

```
Complex z1(1,2), z2(2,3), z3;
cout<<"Number of complexs:"<<Complex::getNumComplex()<<endl;

 cout<<"Number of complexes: "<<z1.getNumComplex()<<endl;
```

non - elegant

OOP: Advanced class features

Complex z1(1,2), z2(2,3), z3;



OOP: Advanced class features

- Classes vs. Structs
  - default access specifier
    - class: `private`
    - struct: `public`
  - class: data + methods, can be used polymorphically
  - struct: mostly data + convenience methods

OOP: Advanced class features

- Classes vs. structures

```
Class list{
private:
  struct node
  {
    node *next;
    int val;
    node( int val = 0, node * next = nullptr):val(val), next(next){}
  };
  node * mHead;
public:
  list ();
  ~list ();
  void insert (int a);
  void printAll()const;
};
```

OOP: Advanced class features

- Passing function arguments
  - by value
    - the function works on a copy of the variable
  - by reference
    - the function works on the original variable, may modify it
  - by constant reference
    - the function works on the original variable, may not modify (verified by the compiler)

OOP: Advanced class features

- Passing function arguments

passing primitive values

```
void f1(int x)  {x = x + 1;}
void f2(int& x) {x = x + 1;}
void f3(const int& x) {x = x + 1;}// !!!!
void f4(int *x) {*x = *x + 1;}

int main(){
    int y = 5;
    f1(y);
    f2(y);
    f3(y);
    f4(&y);
    return 0;
}
```

OOP: Advanced class features

- Passing function arguments

```
void f1(Point p);
void f2(Point& p);
void f3(const Point& p);
void f4(Point *p);

int main(){
    Point p1(3,3);
    f1(p1);
    f2(p1);
    f3(p1);
    f4(&p1);
    return 0;
}
```

copy constructor will be used on the argument

only `const` methods of the class can be invoked on this argument

OOP: Advanced class features

- `friend` functions, `friend` classes, `friend` member functions
  - friends are allowed to access private members of a class
  - Use it rarely
    - operator overloading

OOP: Advanced class features

- `friend` vs. `static` functions

```
class Test{
private:
    int iValue;
    static int sValue;
public:
    Test( int in ):iValue( in ){}
    void print() const;
    static void print( const Test& what );
    friend void print( const Test& what );
};
```

OOP: Advanced class features

- `friend` vs. `static` functions

```
int Test :: sValue = 0;

void Test::print() const{
    cout<<"Member: "<<iValue<<endl;
}

void Test::print( const Test& what ){
    cout<<"Static: "<<what.iValue<<endl;
}

void print( const Test& what ){
    cout<<"Friend: "<<what.iValue<<endl;
}

int main() {
    Test test( 10 );
    test.print();
    Test::print( test );
    print( test );
}
```

OOP: Advanced class features

- `friend` class vs. `friend` member function

```
class List{
private:
     ListElement * head;
public:
     bool find( int key );
     …
};
```

```
class ListElement{
private:
     int key;
     ListElement * next;
     friend class List;
     ...
};
```

```
Class ListElement{
private:
     int key;
     ListElement * next;
     friend class List::find( int key);
     ...
};
```

OOP: Advanced class features

C++03

- Returning a reference to a `const` object

```
// version 1
vector<int> Max(const vector<int> & v1, const vector<int> & v2){
    if (v1.size() > v2.size())
        return v1;
    else
        return v2;
}

// version 2
const vector<int> & Max(const vector<int> & v1, const vector<int> & v2){
    if (v1.size() > v2.size())
        return v1;
    else
        return v2;
}
```

Copy constructor invocation

More efficient

The reference should be to a non-local object

OOP: Advanced class features

C++11

- Returning a reference to a `const` object

```
vector<int> selectOdd( const vector<int>& v){
   vector<int> odds;
   for( int a: v ){
       if (a % 2 == 1 ){
           odds.push_back( a );
       }
   }
   return odds;
}

//...
vector<int> v(N);
for( int i=0; i<N; ++i){
    v.push_back( rand()% M);
}
vector<int> result = selectOdd( v );
```

EFFICIENT! MOVE constructor invocation

OOP: Advanced class features

- Nested classes
  - the class declared within another class is called a *nested class*
  - usually helper classes are declared as nested

```
// Version 1

class Queue
{
 private:
     // class scope definitions
     // Node is a nested structure definition local to this class
     struct Node {Item item; struct Node * next;};
     ...
};
```

OOP: Advanced class features

- Nested classes [Prata]

Node visibility!!!

```
// Version 2

class Queue
{
     // class scope definitions
     // Node is a nested class definition  local to this class
     class Node
     {
     public:
          Item item;
          Node * next;
          Node(const Item & i) : item(i), next(0) { }
     };
     //...
};
```

OOP: Advanced class features

- Nested classes
  - a nested class **B** declared in a **private** section of a class **A**:
    - **B** is local to class **A** (only class A can use it)
  - a nested class **B** declared in a **protected** section of a class **A**:
    - **B** can be used both in **A** and in the derived classes of **A**
  - a nested class **B** declared in a **public** section of a class **A**:
    - **B** is available to the outside world (Usage: `A::B b;`)

OOP: Advanced class features

- Features of a *well-behaved* C++ class
  - implicit constructor
    - `T :: T(){ … }`
  - destructor
    - `T :: ~T(){ … }`
  - copy constructor
    - `T :: T( const T& ){ … }`
  - assignment operator (*see next module*)
    - `T& T :: operator=( const T& ){ … }`

OOP: Advanced class features

- Constructor delegation (C++11)

```
// C++03
class A
{
    void init() { std::cout << "init()"; }
    void doSomethingElse() { std::cout << "doSomethingElse()\n"; }
public:
    A() { init(); }
    A(int a) { init(); doSomethingElse(); }
};
```

```
// C++11
class A
{
    void doSomethingElse() { std::cout << "doSomethingElse()\n"; }
public:
    A() { ... }
    A(int a) : A() { doSomethingElse(); }
};
```

OOP: Advanced class features

- Lvalues:
  - Refer to objects accessible at more than one point in a source code
    - Named objects
    - Objects accessible via pointers/references
  - Lvalues may not be moved from
- Rvalues:
  - Refer to objects accessible at exactly one point in source code
    - Temporary objects (e.g. by value function return)
  - Rvalues may be moved from

OOP: Advanced class features

- Lvalue

```
int  x;
x = 10;
```

Rvalue

OOP: Advanced class features

– Move Semantics (C++11)

```
class string{
    char* data;
public:
   string( const char* );
   string( const string& );
   ~string();
};
```

```
string :: string(const char* p){
    size_t size = strlen(p) + 1;
    data = new char[size];
    memcpy(data, p, size);
}
string :: string(const string& that){
    size_t size = strlen(that.data) + 1;
    data = new char[size];
    memcpy(data, that.data, size);
}

string :: ~string(){
    delete[] data;
}
```

OOP: Advanced class features

- Move Semantics (C++11): lvalue, rvalue

```
string a(x);                                    // Line 1
string b(x + y);                                // Line 2
string c(function_returning_a_string());       // Line 3
```

- **lvalue:** real object having an address
  - **Line 1:** `x`
- **rvalue:** temporary object – no name
  - **Line 2:** `x + y`
  - **Line 3:** `function_returning_a_string()`

OOP: Advanced class features

- Move Semantics (C++11): rvalue reference, move constructor

```
//string&& is an rvalue reference to a string
string :: string(string&& that){
   data = that.data;
   that.data = nullptr;
}
```

- **Move constructor**
  - **Shallow copy** of the argument
  - **Ownership transfer** to the new object

OOP: Advanced class features

- **Move** constructor – Stack class

```
Stack::Stack(Stack&& rhs){
    //move rhs to this
    this->mCapacity = rhs.mCapacity;
    this->mTop = rhs.mTop;
    this->mElements = rhs.mElements;

    //leave rhs in valid state
    rhs.mElements = nullptr;
    rhs.mCapacity = 0;
    rhs.mTop = 0;
}
```

OOP: Advanced class features

- Copy constructor vs. move constructor
  - Copy constructor: **deep copy**
  - Move constructor: **shallow copy + ownership transfer**

```
// constructor
string s="apple";
// copy constructor: s is an lvalue
string s1 = s;
// move constructor: right side is an rvalue
string s2 = s + s1;
```

OOP: Advanced class features

- Passing large objects

```
// C++98
// avoid expense copying

void makeBigVector(vector<int>& out){
 ...
}
vector<int> v;
makeBigVector( v );
```

```
// C++11
// move semantics

vector<int> makeBigVector(){
 …
}
auto v = makeBigVector();
```

- All STL classes have been extended to support **move semantics**
- The content of the temporary created vector is moved in v (not copied)

http://geant4.web.cern.ch/geant4/collaboration/c++11_guidelines.pdf

OOP: Advanced class features

```
class A{
     int value {10};
     static A instance;
public:
     static A& getInstance(){ return instance;}
     static A  getInstanceCopy(){ return instance;}
     int getValue() const { return value;}
     void setValue( int value ){ this->value = value;}
};
```

```
A A::instance;
int main(){
     A& v1 = A::getInstance();
     cout<<"v1: "<<v1.getValue()<<endl;
     v1.setValue(20);
     cout<<"v1: "<<v1.getValue()<<endl;
     A v2 = A::getInstanceCopy();
     cout<<"v2: "<<v2.getValue()<<endl;
     return 0;
}
```

Reference to a static variable → **lvalue**

A temporary copy of instance → **rvalue**

Output?

Module 4

Object-Oriented Programming

Operator overloading

OOP: Operator overloading

Content

- Objectives
- Types of operators
- Operators
  - Arithmetic operators
  - Increment/decrement
  - Inserter/extractor operators
  - Assignment operator (copy and move)
  - Index operator
  - Relational and equality operators
  - Conversion operators

OOP: Operator overloading

Objective

- To make the class usage easier, more intuitive
  - the ability to read an object using the `extractor` operator (>>)
    - `Employee e1; cin >> e;`
  - the ability to write an object using the `inserter` operator (<<)
    - `Employee e2; cout<<e<<endl;`
  - the ability to compare objects of a given class
    - `cout<< ((e1 < e2) ? "less" : "greater");`

Operator overloading: a service to the clients of the class

OOP: Operator overloading

Limitations

- You cannot add new operator symbols. Only the existing operators can be redefined.
- Some operators cannot be overloaded:
  - `.` (member access in an object)
  - `::`(scope resolution operator)
  - **`sizeof`**
  - **`?:`**
- You cannot change the **arity** (the number of arguments) of the operator
- You cannot change the **precedence** or **associativity** of the operator

OOP: Operator overloading

How to implement?

- write a function with the name `operator<symbol>`
- alternatives:
  - method of your class
  - global function (usually a friend of the class)

http://en.cppreference.com/w/cpp/language/operators

OOP: Operator overloading

– There are 3 types of operators:

- operators that must be methods (**member functions**)
  - they don't make sense outside of a class:
    - `operator=, operator(),operator[], operator->`
- operators that must be **global functions**
  - the left-hand side of the operator is a variable of different type than your class: `operator<<, operator>>`
    - **`cout`** `<< emp;`
      - `cout: ostream`
      - `emp: Employee`
- operators that can be **either** methods or global functions
  - **Gregoire:** “Make every operator a method unless you must make it a global function.”

OOP: Operator overloading

- Choosing argument types:
  - value vs. reference
    - Prefer passing-by-reference instead of passing-by-value.
  - `const` vs. non const
    - Prefer const unless you modify it.
- Choosing return types
  - you can specify any return type, however
    - follow the built-in types rule:
      - comparison always return `bool`
      - **arithmetic operators return an object representing the result of the arithmetic**

OOP: Operator overloading

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex{
public:
    Complex(double, double );
    void setRe( double );
    void setIm( double im);
    double getRe() const;
    double getIm() const;
    void print() const;
private:
    double re, im;
};
#endif
```

OOP: Operator overloading

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

Complex::Complex(double re, double im):re(

re),im(im) {} void Complex::setRe( double re){this-

>re = re;}

void Complex::setIm( double im){ this->im =

im;}  double Complex::getRe() const{ return

this->re;}   double Complex::getIm() const{

return this->im;}

void Complex::print()const{    cout<<re<<"+"<<im<<"i";}
```

OOP: Operator overloading

– Arithmetic operators (**member** or standalone func.)
  - unary minus
  - binary minus

```
Complex Complex::operator-() const{
    Complex temp(-this->re, -this->im);
    return temp;
}

Complex Complex::operator-( const Complex& z) const{
    Complex temp(this->re - z.re, this->im- z.im);
    return temp;
}
```

OOP: Operator overloading

- Arithmetic operators (member or **standalone** func.)
  - unary minus
  - binary minus

```
Complex operator-( const Complex& z ){
    Complex temp(-z.getRe(), -z.getIm());
    return temp;
}

Complex operator-( const Complex& z1, const Complex& z2 ){
    Complex temp(z1.getRe()-z2.getRe(), z1.getIm()-z2.getIm());
    return temp;
}
```

OOP: Operator overloading

– Increment/Decrement operators

- postincrement:

  ```
  - int i = 10; int j = i++; // j → 10
  ```

- preincrement:

  ```
  - int i = 10; int j = ++i; // j → 11
  ```

- The C++ standard specifies that the prefix increment and decrement return an **lvalue** (left value).

OOP: Operator overloading

- Increment/Decrement operators (member func.)

```
Complex& Complex::operator++(){            //prefix
      (this->re)++;
      (this->im)++;
      return *this;
}

Complex   Complex::operator++( int ){ //postfix
      Complex
      temp(*this);
      (this->re)++;
      (this-
      >im)++;
      return
      temp;
}
```

Which one is more efficient? Why?

OOP: Operator overloading

- Inserter/Extractor operators (standalone func.)

```
//complex.h

class Complex {
public:
    friend ostream& operator<<(
                          ostream& os, const Complex& c);
    friend istream& operator>>(
                          istream& is, Complex& c);
    //...
};
```

OOP: Operator overloading

- Inserter/Extractor operators (standalone func.)

```
//complex.cpp

ostream& operator<<( ostream& os, const Complex& c){
     os<<c.re<<"+"<<c.im<<"i";
     return os;
}

istream& operator>>( istream& is, Complex& c){
     is>>c.re>>c.im;
     return is;
}
```

OOP: Operator overloading

- Inserter/Extractor operators
- **Syntax:**

```
ostream& operator<<( ostream& os, const T& out)
istream& operator>>( istream& is, T& in)
```

- Remarks:
  - Streams are always *passed by reference*
  - **Q:** Why should inserter operator return an **ostream&**?
  - **Q:** Why should extractor operator return an **istream&**?

OOP: Operator overloading

- Inserter/Extractor operators
- Usage:

```
Complex z1, z2;
cout<<"Read 2 complex number:";
//Extractor
cin>>z1>>z2;
//Inserter
cout<<"z1: "<<z1<<endl;
cout<<"z2: "<<z2<<endl;

cout<<"z1++: "<<(z1++)<<endl;
cout<<"++z2: "<<(++z2)<<endl;
```

OOP: Operator overloading

– **Assignment operator (=)**

- **Q:** When should be overloaded?
- **A:** When bitwise copy is not satisfactory (e.g. if you have dynamically allocated memory ⇒
  - when we should implement the copy constructor and the destructor too).
  - Ex. our Stack class

OOP: Operator overloading

- Assignment operator (member func.)
  - Copy assignment
  - Move assignment (since C++11)

OOP: Operator overloading

- **Copy** assignment operator (member func.)
  - **Syntax:** `X& operator=( const X& rhs);`
  - **Q:** Is the return type necessary?
    - Analyze the following example code

```
Complex z1(1,2), z2(2,3), z3(1,1);
z3 = z1;
z2 = z1 = z3;
```

OOP: Operator overloading

- **Copy** assignment operator example

```
Stack& Stack::operator=(const Stack& rhs) {
 if (this != &rhs) {
   //delete lhs - left hand side
   delete [] this->mElements;
   this->mCapacity = 0;
    this_>melements = nullptr; // in case next line throws
   //copy rhs - right hand side
   this->mCapacity = rhs.mCapacity;
   this->mElements = new double[ mCapacity ];
   int nr = rhs.mTop - rhs.mElements;
   std::copy(rhs.mElements,rhs.mElements+nr,this->mElements);
   mTop = mElements + nr;
 }
 return *this;
}
```

OOP: Operator overloading

- Copy assignment operator vs Copy constructor

```
Complex z1(1,2), z2(3,4); //Constructor
Complex z3 = z1; //Copy constructor
Complex z4(z2);   //Copy constructor
z1 = z2;          //Copy assignment operator
```

OOP: Operator overloading

- **Move** assignment operator (member func.)
  - **Syntax:** `X& operator=( X&& rhs);`
  - When it is called?

```
Complex z1(1,2), z2(3,4); //Constructor
Complex z4(z2); //Copy constructor
z1 = z2;      //Copy assignment operator
Complex z3 = z1 + z2; //Move constructor
z3 = z1 + z1;//Move assignment
```

OOP: Operator overloading

- **Move** assignment operator example

```
Stack& Stack::operator=(Stack&& rhs){
    //delete lhs - left hand side
    delete [] this->mElements;
     //move rhs to this
     this->mCapacity = rhs.mCapacity;
    this->mTop = rhs.mTop;
    this->mElements = rhs.mElements;
    //leave rhs in valid state
    rhs.mElements = nullptr;
    rhs.mCapacity = 0;
     rhs.mTop = 0;
    //return permits s1 = s2 = create_stack(4);
    return *this;
}
```

OOP: Advanced class features

- Features of a *well-behaved* C++ class (**2011**)
  - implicit constructor `T :: T();`
  - destructor `T :: ~T();`
  - copy constructor `T :: T( const T& );`
  - **move** constructor `T :: T( T&& );`
  - copy assignment operator
    - `T& T :: operator=( const T& );`
  - **move** assignment operator
    - `T& T :: operator=( T&& rhs );`

OOP: Operator overloading

- Subscript operator: needed for arrays (member func.)
- Suppose you want your own dynamically allocated C-style array ⇒ implement your own `CArray`

```
#ifndef CARRAY_H
#define   CARRAY_H
class CArray{
public:
    CArray( int size = 10 );
    ~CArray();
    CArray( const CArray&) = delete;
    CArray& operator=( const Carray&) = delete;
    double& operator[]( int index );
    double  operator[]( int index ) const;
private:
    double * mElems;
    int mSize;
};
#endif    /* ARRAY_H */`
```

Provides read-only access

"If the value type is known to be a built-in type, the const variant should return by value."
http://en.cppreference.com/w/cpp/language/operators.

OOP: Operator overloading

- Implementation

```
CArray::CArray( int size ){ if( size
    < 0 ){
        this->size = 10;
    }
    this->mSize = size;
    this->mElems = new double[ mSize ];
}

CArray::~CArray(){
    if( mElems != nullptr ){
        delete[] mElems;  mElems =
        nullptr;
    }
}

double& CArray::operator[]( int index ){ if(
    index <0 || index >= mSize ){
        throw out_of_range("");
    }
    return mElems[ index ];
}
```

```
double CArray::operator[](
                    int index ) const{
    if( index <0 || index >= mSize ){
        throw out_of_range("");
    }
    return mElems[ index ];
}
```

#include<stdexcept>

OOP: Operator overloading

- `const` vs non-const `[]` `operator`

```
void printArray(const CArray& arr, size_t size) {
    for (size_t i = 0; i < size; i++) {
        cout << arr[i] << "" ;
        // Calls the const operator[] because arr is
        // a const object.
    }
    cout << endl;
}
```

```
CArray myArray;
for (size_t i = 0; i < 10; i++) {
    myArray[i] = 100;
    // Calls the non-const operator[] because
    // myArray is a non-const object.
}
printArray(myArray, 10);
```

OOP: Operator overloading

- Relational and equality operators
  - used for **search** and **sort**
  - the container must be able to compare the stored objects

```
bool operator ==( const Point& p1, const Point& p2){
    return p1.getX() == p2.getX() && p1.getY() == p2.getY();
}


bool operator <( const Point& p1, const Point& p2){
   return p1.distance(Point(0,0)) < p2.distance(Point(0,0));
}
```

```
set<Point> p;
```

```
vector<Point> v; //...
sort(v.begin(), v.end());
```

OOP: Operator overloading

- The function call operator `()`
- Instances of classes overloading this operator behave as functions too (they are function objects = function + object)

```
#ifndef ADDVALUE_H
#define   ADDVALUE_H
class AddValue{
    int value;
public:
    AddValue( int inValue = 1);
    void operator()( int& what );
};
#endif    /* ADDVALUE_H */
```

```
#include "AddValue.h"

AddValue::AddValue( int inValue ){
    this->value = inValue;
}

void AddValue::operator()( int& what ){
    what += this->value;
}
```

OOP: Operator overloading

- The function call operator

```
AddValue func(2);
int array[]={1, 2, 3};
for( int& x : array ){
   func(x);
}
for( int x: array ){
   cout <<x<<endl;
}
```

OOP: Operator overloading

- Function call operator
  - used frequently for defining sorting criterion

```
struct EmployeeCompare{
    bool operator()( const Employee& e1, const Employee&
        e2){ if( e1.getLastName() == e2.getLastName())
            return e1.getFirstName() < e2.getFirstName();
        else
            return e1.getLastName() < e2.getLastName();

    }
};
```

OOP: Operator overloading

- Function call operator
  - sorted container

```
set<Employee, EmployeeCompare> s;

Employee e1; e1.setFirstName("Barbara");
e1.setLastName("Liskov");
Employee e2; e2.setFirstName("John");
e2.setLastName("Steinbeck");
Employee e3; e3.setFirstName("Andrew");
e3.setLastName("Foyle");
s.insert( e1 ); s.insert( e2 ); s.insert( e3 );

for( auto& emp : s){
   emp.display();
}
```

OOP: Operator overloading

- Sorting elements of a given *type*:
  - **A.** override operators: **<**, **==**
  - **B.** define a function object containing the comparison
- **Which one to use?**
  - **Q:** How many sorted criteria can be defined using method **A**?
  - **Q:** How many sorted criteria can be defined using method **B**?

OOP: Operator overloading

- Writing conversion operators

```
class Complex{
public:
  operator string() const;
  //
};

Complex::operator string() const{
   stringstream ss;
   ss<<this->re<<"+"<<this->im<<"i";
   return ss.str();
}
```

```
//usage
Complex z(1, 2);
string a = z;
cout<<a<<endl;
```

OOP: Operator overloading

- After templates
  - Overloading operator *
  - Overloading operator →

OOP: Review

- Find all possible errors or shortcommings!

```
class Array {
public:
  Array (int n) : rep_(new int [n]) { }
  Array (Array& rhs) : rep_(rhs.rep_) { }
  ~Array () { delete rep_; }
  Array& operator = (Array rhs) { rep_= rhs.rep_;  }
  int& operator [] (int n) { return &rep_[n]; }
private:
  int * rep_;
}; // Array
```

Source: http://www.cs.helsinki.fi/u/vihavain/k13/gea/exer/exer_2.html

Solution required!

- It is given the following program!

```
#include <iostream>

int main(){
    std::cout<<"Hello\n";
    return 0;
}
```

Modify the program *without modifying the main function* so that the output of the program would be:

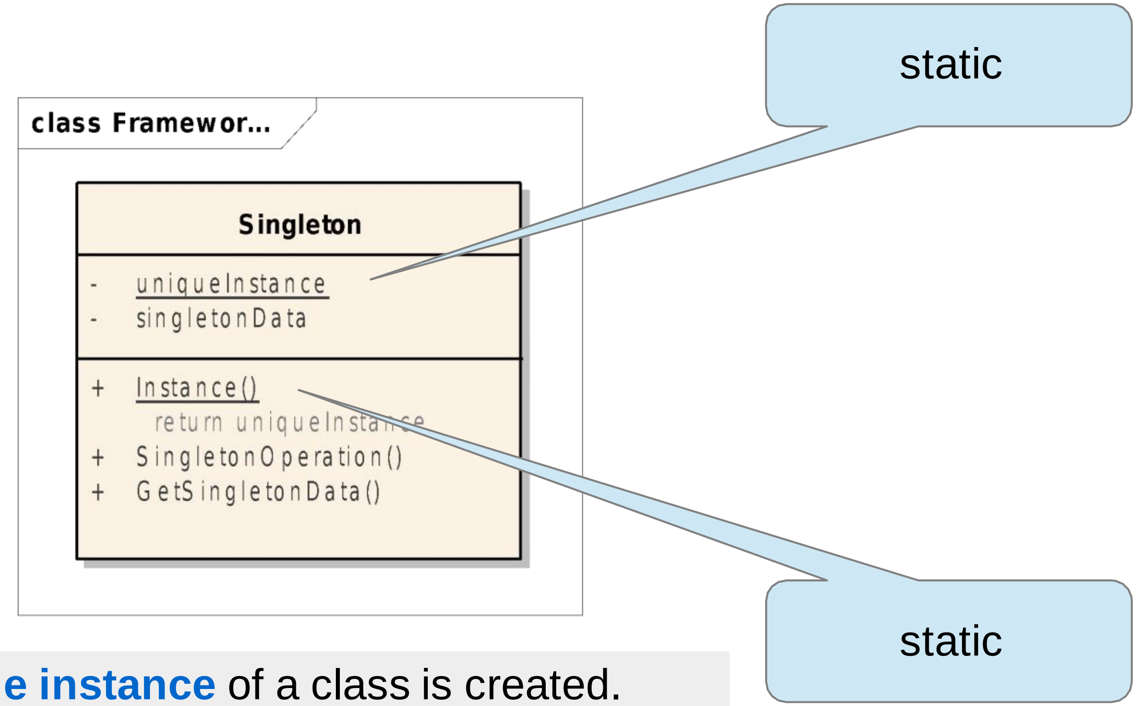
```
Start
Hello
Stop
```

Singleton Design Pattern

```
#include <string>
class Logger{
public:
    static Logger* Instance();
    bool openLogFile(std::string logFile);
    void writeToLogFile();
    bool closeLogFile();
private:
    Logger(){};  // Private so that it can  not be called
    Logger(Logger const&){}; // copy constructor is private
    Logger& operator=(Logger const&){};// assignment operator is private
    static Logger* m_pInstance;
};
```

http://www.yolinux.com/TUTORIALS/C++Singleton.html

Singleton Design Pattern



- Ensure that **only one instance** of a class is created.
- Provide a **global point of access** to the object.

Module 5

Object-Oriented Programming

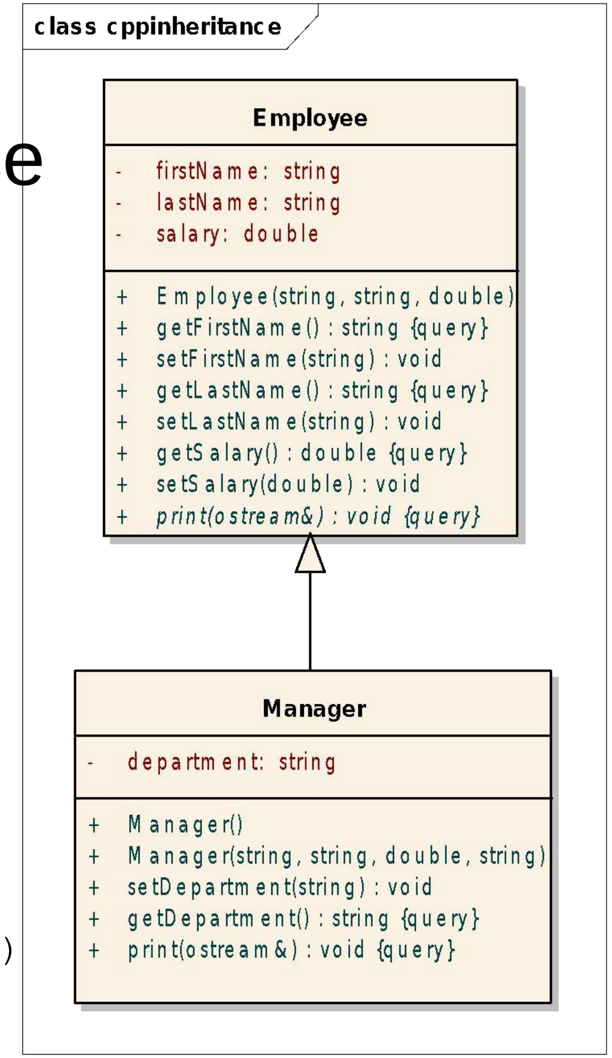
Public Inheritance

OOP: Inheritance

- Inheritance
  - *is-a* relationship - public inheritance
  - protected access
  - virtual member function
  - early (static) binding vs. late (dynamic) binding
  - abstract base classes
  - pure virtual functions
  - virtual destructor

OOP: Inheritance

- `public` inheritance
  - *is-a* relationship
  - base class: Employee
  - derived class: Manager
- You can do with inheritance
  - *add data*
    - ex. `department`
  - *add functionality*
    - ex. `getDepartment()`, `setDepartment()`
  - *modify methods' behavior*
    - ex. `print()`



OOP: Inheritance

- `protected` access
  - base class's private members can not be accessed in a derived class
  - base class's protected members can be accessed in a derived class
  - base class's public members can be accessed from anywhere

OOP: Inheritance

- `public` inheritance

```
class Employee{
public:
    Employee(string firstName = "", string lastName = "",
             double salary = 0.0) : firstName(firstName),
                                    lastName(lastName),
                                    salary(salary) {
    }
    //...
};
```

```
class Manager:public Employee{
    string department;
public:
    Manager();
    Manager( string firstName, string lastName, double salary,
             string department );
    //...
};
```

OOP: Inheritance

– Derived class's constructors

```
Manager::Manager(){
}
```

Employee's constructor invocation → Default constructor can be invoked implicitly

OOP: Inheritance

- Derived class's constructors

```
Manager::Manager(){
}
```

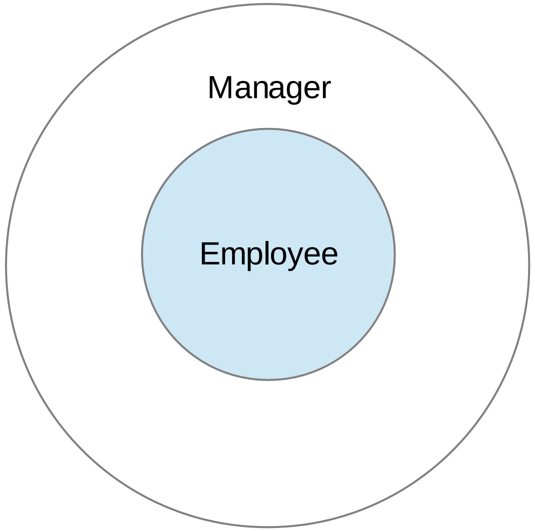
Employee's constructor invocation → Default constructor can be invoked implicitly

```
Manager::Manager(string firstName, string lastName, double salary,
                 string department):Employee(firstName, lastName, salary),
                                    artment(department){


}
```

base class's constructor invocation – *constructor initializer list*
arguments for the base class's constructor are specified in the definition of a derived class's constructor

OOP: Inheritance

- How are derived class's objects constructed?
  - *bottom up* order:
    - base class constructor invocation
    - member initialization
    - derived class's constructor block
  - destruction
    - in the opposite order



OOP: Inheritance

- Method overriding

```
class Employee{
public:
     virtual void print(ostream&) const;
};
```

```
class Manager:public Employee{
public:

    virtual void print(ostream&) const;
};
```

OOP: Inheritance

- Method overriding

```
class Employee {
public:
     virtual void print( ostream&) const;
};
```

```
void Employee::print(ostream& os ) const{
    os<<this->firstName<<" "<<this->lastName<<" "<<this->salary;
}
```

```
class Manager:public Employee{
public:
    virtual void print(ostream&) const;
};
```

```
void Manager::print(ostream& os) const{
    Employee::print(os);
    os<<" "<<department;
}
```

OOP: Inheritance

- Method overriding - virtual functions
  - non virtual functions are bound statically
    - compile time
  - virtual functions are bound dynamically
    - run time

OOP: Inheritance

- Polymorphism

```
void printAll( const vector<Employee*>& emps ){
    for( int i=0; i<emps.size(); ++i){
        emps[i]-> print(cout);
        cout<<endl;
    }
}

int main(int argc, char** argv) {
    vector<Employee*> v;
    Employee e("John", "Smith", 1000);
    v.push_back(&e);
    Manager m("Sarah", "Parker", 2000, "Sales");
    v.push_back(&m);
    cout<<endl;
    printAll( v );
    return 0;
}
```

Output:

```
John Smith 1000
Sarah Parker 2000 Sales
```

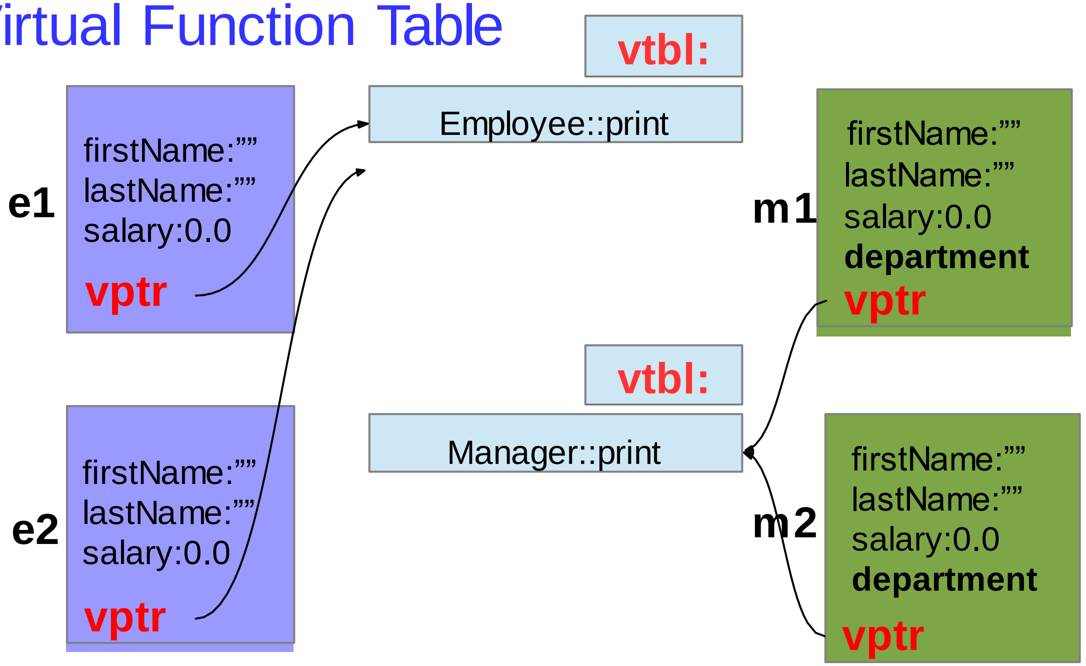
OOP: Inheritance

- Polymorphism
  - a type with virtual functions is called a polymorphic type
  - polymorphic behavior **preconditions**:
    - the member function must be virtual
    - objects must be manipulated through
      - pointers or
      - references
    - `Employee :: print( os )` static binding – no polymorphism

OOP: Inheritance

– Polymorphism – Virtual Function Table

```
class Employee{
public:
 virtual void print(ostream&) const;
     //...
};

class Manager:public Employee{
 virtual void print(ostream&) const;
     //...
};
Employee e1, e2;
Manager m1, m2;
```



Discussion!!!

```
Employee * pe;
pe = &e1; pe->print(); //???
pe = &m2; pe->print(); //???
```

Each class with virtual functions has its own virtual function table (vtbl).

RTTI – Run-Time Type Information

`dynamic_cast<>`(pointer)

```
class Base{};
class Derived : public Base{};

Base* basePointer = new Derived();
Derived* derivedPointer = nullptr;

//To find whether basePointer is pointing to Derived type of object

derivedPointer =
dynamic_cast<Derived*>(basePointer); if
(derivedPointer != nullptr){
   cout << "basePointer is pointing to a Derived class object";
}else{
   cout << "basePointer is NOT pointing to a Derived class object";
}
```

Java:
instanceof

RTTI – Run-Time Type Information

`dynamic_cast<>`(reference)

```
class Base{};
class Derived : public Base{};

Derived derived;
Base& baseRef = derived;

// If the operand of a dynamic_cast to a reference isn't of the expected type,
//  a bad_cast exception is thrown.

try{
       Derived& derivedRef = dynamic_cast<Derived&>(baseRef);
} catch( bad_cast ){
       // ..
}
```

OOP: Inheritance

- Abstract classes
  - used for representing abstract concepts
  - used as base class for other classes
  - no instances can be created

OOP: Inheritance

- Abstract classes – pure virtual functions

```
class Shape{ // abstract class
    public:
        virtual void rotate(int)  = 0;      // pure virtual function
        virtual void draw()  = 0;  // pure virtual function
        // ...
};
```

```
 Shape s; //???
```

OOP: Inheritance

- Abstract classes – pure virtual functions

```
class Shape{ // abstract class
    public:
        virtual void rotate(int)  = 0;      // pure virtual function
        virtual void draw()  = 0;  // pure virtual function
        // ...
};
```

```
Shape s; //Compiler error
```

OOP: Inheritance

- Abstract class → concrete class

```
class Point{ /* ... */ };
class Circle : public Shape {
    public:
        void rotate(int);          // override Shape::rotate
        void draw();               // override Shape::draw
        Circle(Point p, int r) ;
    private:
        Point center;
        int radius;
};
```

OOP: Inheritance

- Abstract class → abstract class

```
class Polygon : public Shape{
public:
    // draw() and rotate() are not overridden

};
```

```
Polygon p; //Compiler error
```

OOP: Inheritance

- Virtual destructor
  - Every class having at least one virtual function should have virtual destructor. Why?

```
class X{
public:
    // ...
    virtual ~X();
};
```

OOP: Inheritance

- Virtual destructor

```
void deleteAll( Employee ** emps, int size){
    for( int i=0; i<size; ++i){
        delete emps[ i ];
    }
    delete [] emps;
}

 // main
 Employee ** t = new Employee *[ 10 ];
 for(int i=0; i<10; ++i){
    if( i % 2 == 0 )
        t[ i ] = new Employee();
    else
        t[ i ] = new Manager();
 }
 deleteAll( t, 10);
```

Which destructor is invoked?

Module 6

Object-Oriented Programming

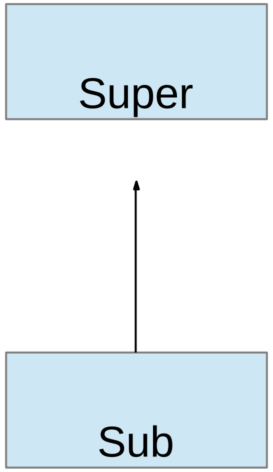
Object relationships

OOP: Object relationships

- The ***is-a*** relationship
  - Private inheritance
  - Multiple inheritance

- The ***has-a*** relationship
  - Association
    - Composition (strong containment)
    - Aggregation (weak containment)

OOP: Object relationships



- The *is-a* relationship – *Client's view (1)*
  - works in only *one direction:*
    - every **Sub** object **is** also **a Super** one
    - but **Super** object **is not a Sub**

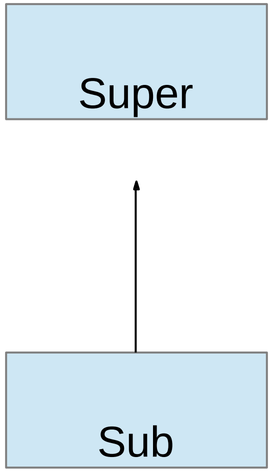
```
void foo1( const Super& s );
void foo2( const Sub& s);
Super super;
Sub sub;

foo1(super); //OK
foo1(sub);    //OK
foo2(super); //NOT OK
foo2(sub);   //OK
```

OOP: Object relationships

- The *is-a* relationship – *Client's view (2)*

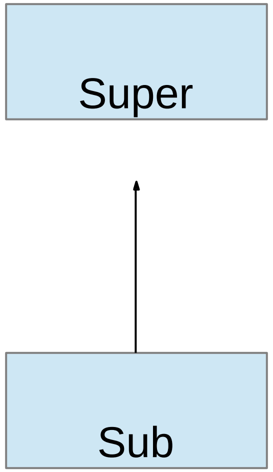
```
class Super{
public:
   virtual void method1();
};
class Sub : public Super{
public:
   virtual void method2();
};
```

```
Super * p= new Super();
p->method1(); //OK

p = new Sub();
p->method1(); //OK
p->method2(); //NOT OK
((Sub *)p)->method2();//OK
```



OOP: Object relationships

- The *is-a* relationship – *Sub-class's view*



- the `Sub` class augments the `Super` class by **adding additional methods**
- the `Sub` class **may override** the `Super` class **methods**
- the subclass can use all the public and protected members of a superclass.

OOP: Object relationships

- The *is-a* relationship: *preventing inheritance* ***C++11***
  - `final` classes – cannot be extended

```
class Super final
{

};
```

OOP: Object relationships

- The *is-a* relationship: *a client's view of overridden methods(1)*
  - *polymorphism*

```
class Super{
public:
    virtual void method1();
};
class Sub : public Super{
public:
    virtual void method1();
};
```

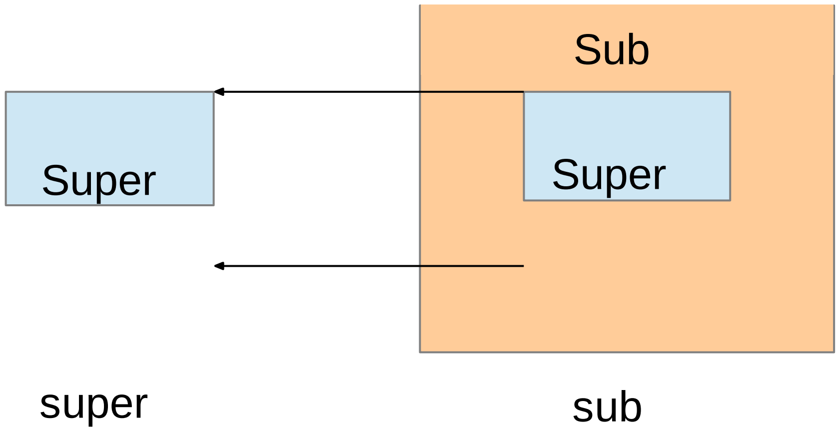
```
Super super;
super.method1(); //Super::method1()

Sub sub;
sub.method1();   //Sub::method1()

Super& ref =super;
ref.method1();   //Super::method1();

ref = sub;
ref.method1();   //Sub::method1();

Super* ptr =&super;
ptr->method1(); //Super::method1();

ptr = &sub;
ptr->method1();   //Sub::method1();
```

OOP: Object relationships

- The *is-a* relationship: *a client's view of overridden methods(2)*
  - object slicing

```
class Super{
public:
    virtual void method1();
};
class Sub : public Super{
public:
    virtual void method1();
};
```

```
Sub sub;
Super super = sub;
super.method1(); // Super::method1();
```



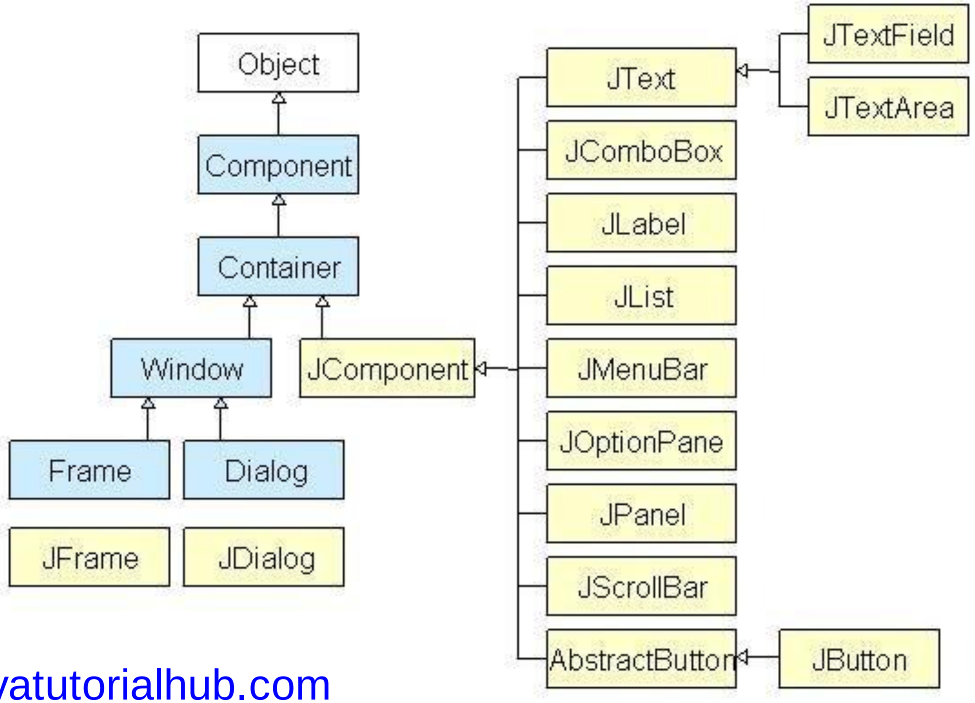
OOP: Object relationships

- The *is-a* relationship: *preventing method overriding* ***C++11***

```
class Super{
public:
    virtual void method1() final;
};
class Sub : public Super{
public:
    virtual void method1(); //ERROR
};
```

OOP: Object relationships

- Inheritance for polymorphism



www.javatutorialhub.com

OOP: Object relationships

- The *has-a* relationship

OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**

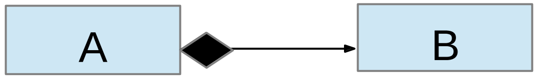
```
class B;

class
A{
private
:
    B b;
};
```

```
class B;

class
A{
private
:
    B*  b;
};
```

```
class B;

class
A{
private
:
    B&  b;
};
```

OOP: Object relationships

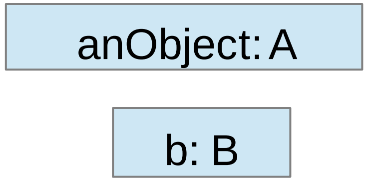
- Implementing the *has-a* relationship



  - An object **A** has an object **B**
    - **strong containment (composition)**

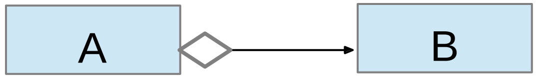
```
class B;

class A{
private:
    B b;
};
```

```
A anObject;
```



OOP: Object relationships

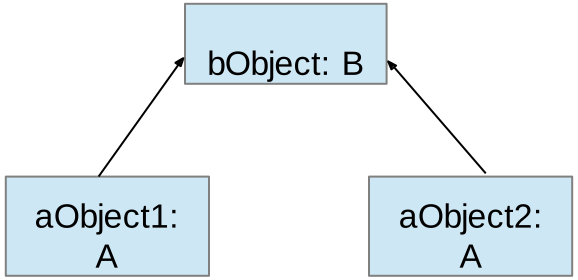
– Implementing the *has-a* relationship



- An object **A** has an object **B**
  - **weak containment (aggregation)**

```
class B;

class A{
private:
    B& b;
public:
    A( const B& pb):b(pb){}
};
```

```
B bObject;
A aObject1(bObject);
A aObject2(bObject);
```



OOP: Object relationships

- Implementing the *has-a* relationship
  - An object **A** has an object **B**

weak containment

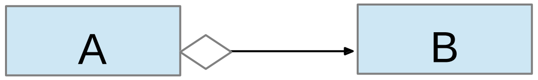
```
class B;

class A{
private:
     B* b;
public:
     A( B* pb):b( pb ){}
};
```

strong containment

```
class B;

class A{
private:
     B* b;
public:
     A(){
          b = new B();
     }
     ~A(){
          delete b;
     }
};
```

OOP: Object relationships

- Implementing the *has-a* relationship

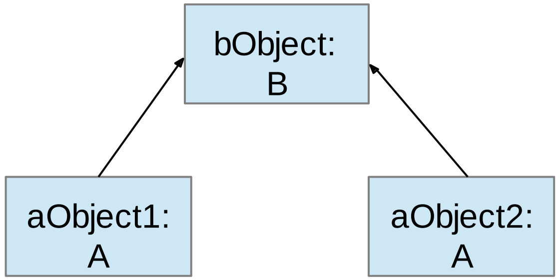


  - An object **A** has an object **B**

weak containment

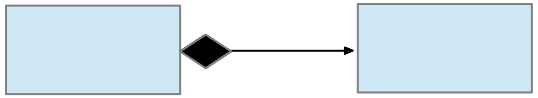
```
class B;

class A{
private:
    B* b;
public:
    A( B* pb):b( pb ){}
};
```

```
Usage:
    B bObject;
    A aObject1(&bObject);
    A aObject2(&bObject);
```



OOP: Object relationships

– Implementing the *has-a* relationship



A

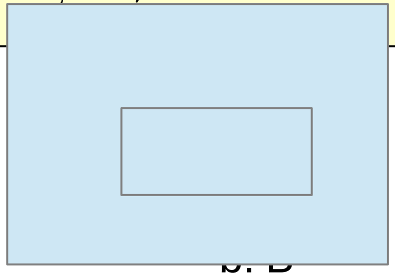
B

- An object **A** has an object

B strong containment

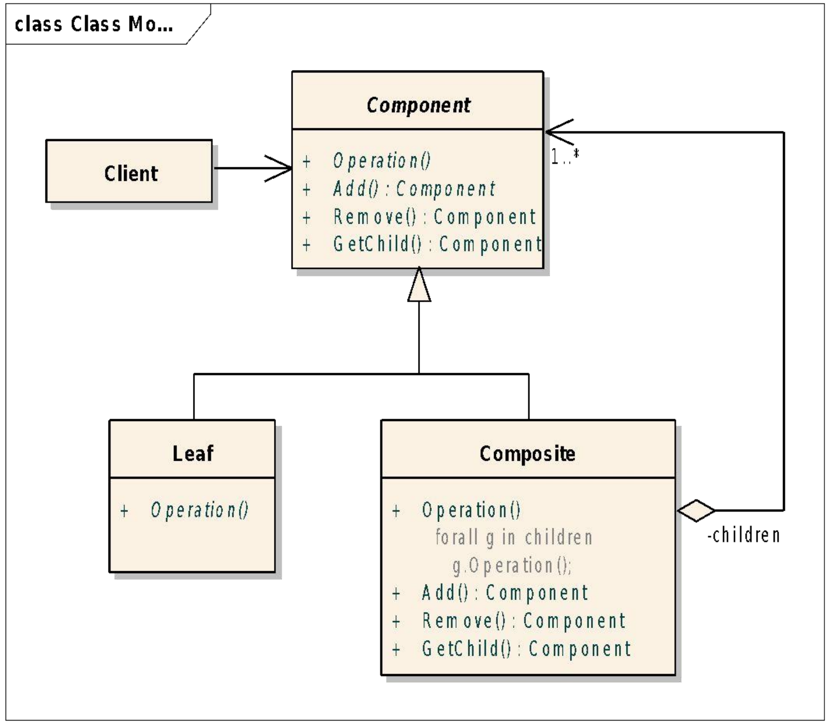
```
class B;

class
A{
priva
te:
    B*
b;
public
:
    A(){
        b = new B();
    }
    ~A(){
        delete b;
```

```
Usage:
    A aObject;
```

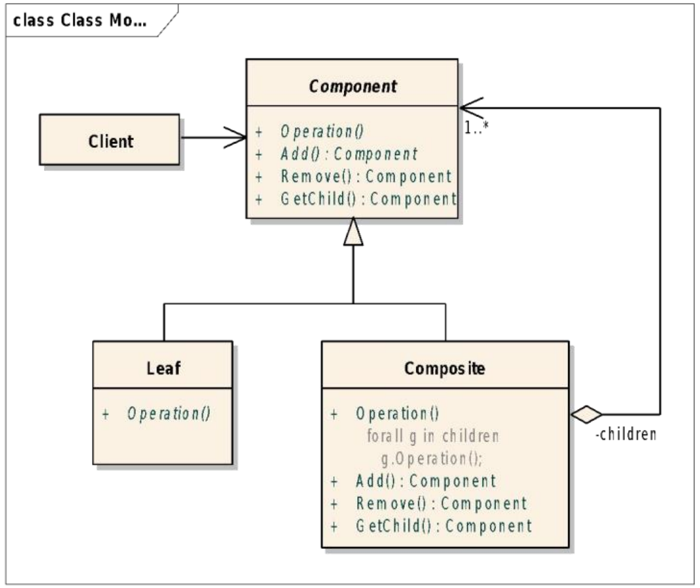


OOP: Object relationships

- Combining the *is-a* and the *has-a* relationships



Composite Design Pattern



- Compose objects into tree structures to represent **part-whole hierarchies.**
- Lets clients treat **individual objects** and **composition of objects uniformly.**

Composite Design Pattern

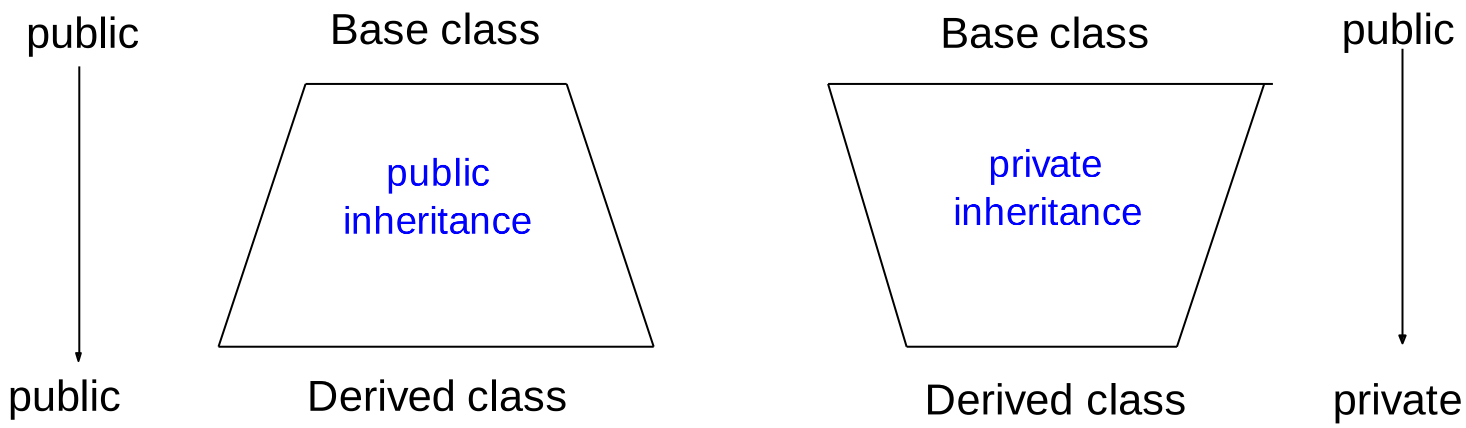
Examples:

- Menu – MenuItem: Menus that contain menu items, each of which could be a menu.
- Container – Element: Containers that contain Elements, each of which could be a Container.
- GUI Container – GUI component: GUI containers that contain GUI components, each of which could be a container

Source: http://www.oodesign.com/composite-pattern.html

Private Inheritance

– another possibility for *has-a* relationship



Derived class **inherits** the base class behavior

Derived class **hides** the base class behavior

Private Inheritance

```
template <typename T>
class MyStack : private vector<T> {
public:
    void push(T elem) {
        this->push_back(elem);
    }
    bool isEmpty() {
        return this->empty();
    }
    void pop() {
        if (!this->empty())this->pop_back();
    }
    T top() {
        if (this->empty()) throw out_of_range("Stack is empty");
        else return this->back();
    }
};
```

Why is **public inheritance** in this case dangerous???

Non-public Inheritance

- it is very rare;
- use it cautiously;
- most programmers are not familiar with it;

What does it print?

```
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
            cout<<"Super"<<endl;
     }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d){
            cout<<"Sub"<<endl;
     }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

What does it print?

```
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
            cout<<"Super"<<endl;
     }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d){
            cout<<"Sub"<<endl;
     }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

creates a new method, instead of overriding the method

The `override` keyword C++11

```
class Super{
public:
    Super(){}
    virtual void someMethod(double d) const{
            cout<<"Super"<<endl;
     }
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double d) const override{
            cout<<"Sub"<<endl;
     }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```