CSC230

Intro to C++ Lecture 5

Outline

- Lab 3 discussion
- Classes and Objects
- Dynamic memory allocation
- Object initialization

Lab 3 section 1 discussion

Lab 3 section 1

- Reverse the word

- Take the user input and index
- Store the input into a array / vector
- Remove the element with the index (optional)
  - vec.earse(vec.begin()+1)
- Output the reversed word

Lab 3 discussion

- Lab 3 section 2
  - Insert an element into a 2D Vector
  - Why we cannot use 2D Array ?
  - Reverse the new 2D Vector
    - Hint: Build a function to reverse the 1d vector
    - Feed this function with each row of the 2D vector

Outline

- Lab 3 discussion
- Classes and Objects
- Dynamic memory allocation
- Object initialization

Class and Object


```
class Base {

  public:

  // public members go here

  protected:

  // protected members go here

  private:

  // private members go here

};
```

- Access specifiers: public, private, protected
- Each class may have multiple sections
- Each section remains effective until either another section or the end of the class body
- The default access is private

Class and object example


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

class student
{
  public:
    char  name[50];
    char  major[50];
    char  homeAddress[100];
};

int main ()
{
  student csStudent, mathStudent;
  strcpy(csStudent.name, "Mike Lee");
  strcpy(csStudent.major, "CS");
  strcpy(csStudent.homeAddress, "Earth");
  cout << csStudent.name << " " << csStudent.homeAddress <<endl;
  return 0;
}
```

Method definition


```
class employee
{
  public:
      ...
    int   id;

    int getID(){
      return id;
    }
};
```

```
class employee
{
  public:
      ...
    int   id;

    int getID();
};

int employee::getID(){
  return id;
}
```

declaration

definition

scope operator

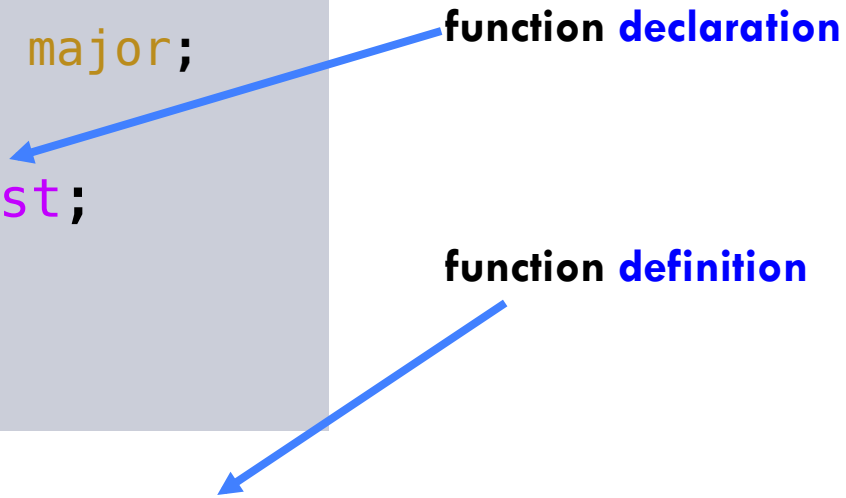
Const method/member function

- **const** method/member function
  - declaration
    - *return_type func_name* (*para_list*) **const;**
  - definition
    - *return_type func_name* (*para_list*) **const** { … }
    - *return_type class_name* :: *func_name* (*para_list*) **const** { … }
  - It is **illegal** for a const member function to **modify** a class data member

Example: const_keyword.cpp

Const Member Function


```
class  student
{
 private :
  string name, addr, major;
 public :
    void info() const;
};
```



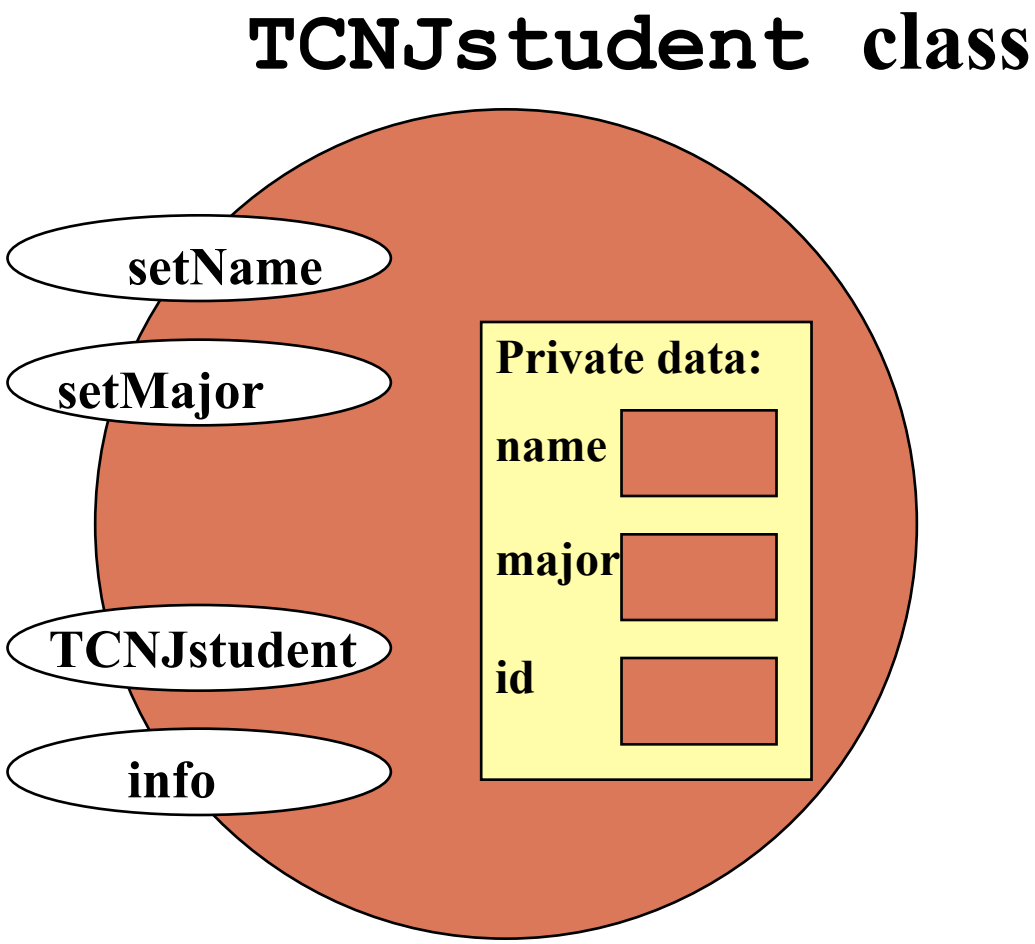

```
void student:: info( ) const
{
  cout << name << ":" << addr << ":" << major << endl;
}
```

TCNJstudent class


```
class TCNJstudent
{
  private:
    char  name[50];
    char  major[50];
    int   id;
    TCNJstudent();
  public:
    void setName();
    void setMajor();
    void info() const;
};
```

Bugs?

Class Interface



Access specifier

Access From	Public	Protected	Private
Same class	Yes	Yes	Yes
Derived classes	Yes	Yes	No
Everywhere	Yes	No	No

- The **default** access specifier is private.
- The data members are usually private or protected. A private member can be accessed by another member function of the same class (exception friend function, more details later)
- Each access control section is optional, repeatable, and sections may occur in any order

One more example


```
#include <iostream>
class circle
{
   private:
     double radius;
   public:
     void store(double);
     double area(void);
     void display(void);
};
```

```
// member function definitions
void circle::store(double r)
{
  radius = r;
}

double circle::area(void)
{
  return 3.14*radius*radius;
}

void circle::display(void)
{
  std::cout << "r = " << radius << std::endl;
}
```

```
int main(void) {
  circle c;   // an object of circle class
  c.store(5.0);
  std::cout << "The area of circle c is " << c.area() << std::endl;
  c.display();
}
```

Look inside the example


```
int main(void) {
  circle c;   // an object of circle class
  c.store(5.0);
  std::cout << "The area of circle c is " << c.area() << std::endl;
  c.display();
}
```

c is **statically** allocated

endl is defined in std **namespace**

Does this one work?


```
int main(void) {
  circle c, *d;
  d.store(5.0);
  std::cout << "The area of circle c is " << d.area() << std::endl;
  d.display();
}
```

- d is a pointer, which should have the address of someone in the memory.
- Did we initialize d ? NO!
- Did compile initialize it? NO!

First modification


```
int main(void) {
  circle c, *d;
  d = &c;
  d.store(5.0);
  std::cout << "The area of circle c is " << d.area() << std::endl;
  d.display();
}
```

- d is initialized
- d is a pointer, we **cannot** use d.store(), d.area(), d.display() to access the functions.

Second modification

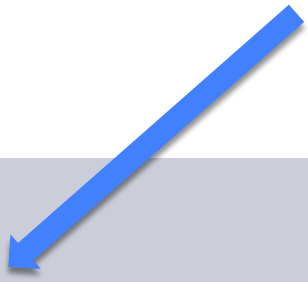

```
int main(void) {
  circle c, *d;
  d = &c;
  d->store(5.0);
  std::cout << "The area of circle c is " << d->area() << std::endl;
  d->display();
}
```

Outline

- Lab 3 discussion
- Classes and Objects
- Dynamic memory allocation
- Object initialization

Pointer, dynamic memory

d is **dynamically** allocated



```
int main(void) {
  circle *d;
  d = new circle();
  d->store(5.0);
  std::cout << "The area of circle c is " << d->area() << std::endl;
  d->display();
}
```

Pointer, dynamic memory

- Compile Time Allocation
  - Memory for named variables is allocated by the compiler
  - Exact size and type of storage must be known at compile time
  - For standard array declarations, this is why the size has to be constant

- Dynamic Memory Allocation
  - Memory allocated "on the fly" during run time
  - Exact amount of space or number of items does not have to be known by the compiler in advance.
  - For dynamic memory allocation, pointers are crucial

  - Example -- test_dynamic.cpp

More example of dynamic memory

Method 1 (using new)

- Allocates memory for the object (or global variables) on the **heap (controlled by programmer)**.
- Requires you to explicitly delete your object later. (If you don't delete it, you could create a memory leak)
- Memory stays allocated until you delete it. (i.e. you could return an object that you created using new)
- It should always be deleted, regardless of which control path is taken, or if exceptions are thrown.

Method 2 (not using new)

- Allocates memory for the object on the **stack (controlled by compiler, where all local variables go)**
- Less memory for stack, easy to get stack overflow
- You don't need to delete it.
- Memory is no longer allocated when it goes out of scope
- **Example – test_stack_heap.cpp**

More example of dynamic memory

d and e are **dynamically** allocated

```
int main(void) {
  circle *d;
  d = new circle();
  int * e = new int[15];
  d->store(5.0);
  std::cout << "The area of circle c is " << d->area() << std::endl;
  d->display();
  delete d;
  delete[] e;
}
```

Memory of d and e are **released** back to the system

```
delete release one element
delete [] release one array
```

You need to call delete or delete [] as many times you called new or new [] respectively.

Outline

- Lab 3 discussion
- Classes and Objects
- Dynamic memory allocation
- Object initialization

Static vs. Non-static

non-static data member

Each object has its own copy

static data member

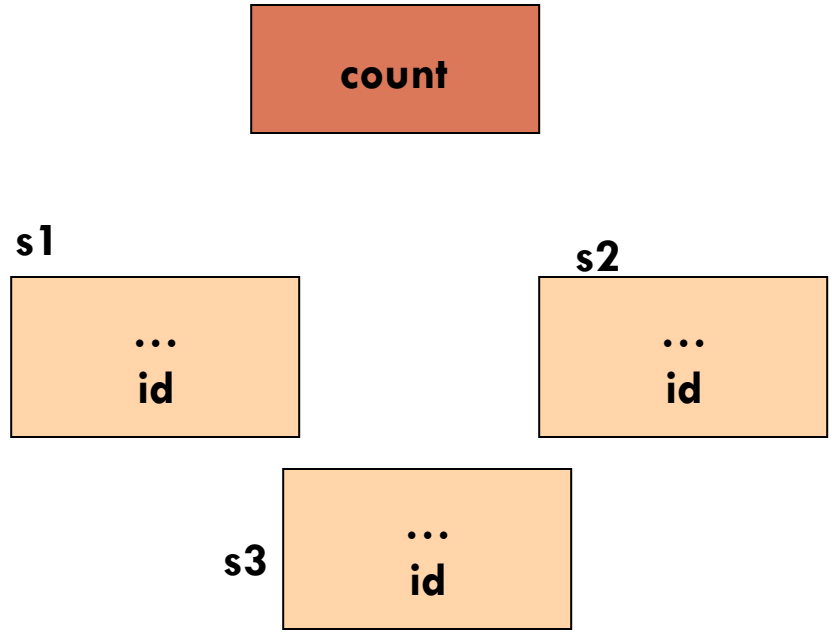
One copy per class type, e.g. counter

```
employee s1;
employee s2;
employee s3;
```

```
class employee
{
  public:
      ...
    int   id;
    static int counter;

    int getID(){
      return id;
    }
};
```

- Examples – static_keyword



Object initialization


```
#include <iostream>

class circle
{
   private:
     double radius;

   public:
     void set(double r);
};
// member function definitions

void circle::set(double r)
{
  radius = r;
}
```

```
int main(void) {
  circle *d;
  d = new circle();
  d->set(5.0);

  circle c;
  c.set(4.0);
}
```

Object initialization, Constructor


```
class circle
{
   private:
     double radius;

   public:
     void set(double r);
     circle();
     circle(const circle &r);
     circle(double r);
};
```

- Default constructor
- Copy constructor
- Constructor with parameters

- **Publicly accessible**
- **same name as the class**
- **no return type**
- **to initialize class data members**
- **different signatures**

Object initialization, Constructor


```
class circle
{
   private:
     double radius;

   public:
     void set(double r);
};
```

When a class is declared with **no constructors**,
the compiler **automatically** assumes default constructor and **copy constructor** for it.

- **Default constructor**

```
circle:: circle() { };
```

- **Copy constructor**

```
circle:: circle (const circle & r)
{
  radius = r.radius;
};
```

Object initialization, Constructor


```
class circle
{
   private:
     double radius;

   public:
     void set(double r);
};
```

- Initialize with default constructor

```
circle r1;
circle *r2 = new circle();
```

- Initialize with copy constructor

•Copy constructor is called when a new object is created from an existing object

•Assignment operator is called when an already initialized object is assigned a new value from another existing object.

If no customer defined constructors. C++ provides default constructors and copy constructor.

Let's check the example, test_copy.cpp

```
circle r3;          //default
r3.set(5.0);

circle r4 = r3;     //copy
circle r5(r4);      //copy

circle *r6 = new circle(r4); //copy
```

Object initialization, Constructor


```
class circle
{
   public:
     double radius;

   public:
     void set(double r);

     circle(double r){radius = r;}

};
```

If **any** constructor is declared,

- no default constructor will exist, unless you define it.
- still have copy constructor

```
circle r1;
```

✗

- Initialize with constructor

```
circle r1(5.0);
circle *r2 = new circle(6.0);
```

✔

Constructor and destructor

An object can be initialized by

- Default constructor
- Copy constructor
- Constructor with parameters

When the object is initialized, resources are allocated.

Just before the object is terminated, the allocated resources should be returned to system.

Destructor


```
class account
{
  private:
    char *name;
    double balance;
    unsigned int id;

  public:
    account();
    account(const account &c);
    account(const char *d);
    ~account();
}

account::~account()
{
  delete[] name;
}
```

destructor:

- Its name is class name preceded by ~
- No argument
- Release dynamic memory and cleanup
- Automatically executed before object goes

out of scope, or when delete a pointer to a object.

Destructor declaration

Destructor definition

Delete whole string.

```
delete name;
```

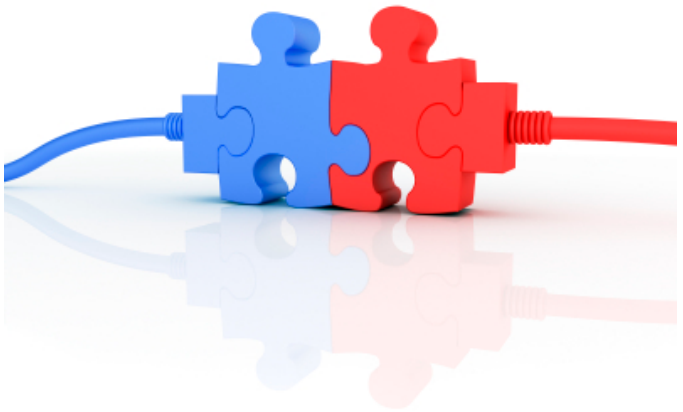
Delete one char.

Work with multiple files

A set of .cpp and .h files for each class group

- .h file contains the prototype of the class
- .cpp contains the implementation of the class

A .cpp file containing the main() function should include all the corresponding .h files where the functions used in .cpp file are declared.



Example: TCNJstudent.h


```
class TCNJstudent
{
  private:
    char  name[50];
    char  major[50];
    int   id;
  public:
    void setName();
    void setMajor();
    TCNJstudent();
    void info() const;
};
```

Example: TCNJstudent.cpp


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

void TCNJstudent::setName()
{ …
}
void TCNJstudent::setMajor()
{ …
}
TCNJstudent::TCNJstudent()
{ …
}
void TCNJstudent::info()
{ …
}
```

Assume the implementation needs this file

Must include the corresponding header file

To simplify the example, we use blank body. A real implementation can have various body.

Example: main.cpp


```
#include "TCNJstudent.h"

int main(){
    …
}
```

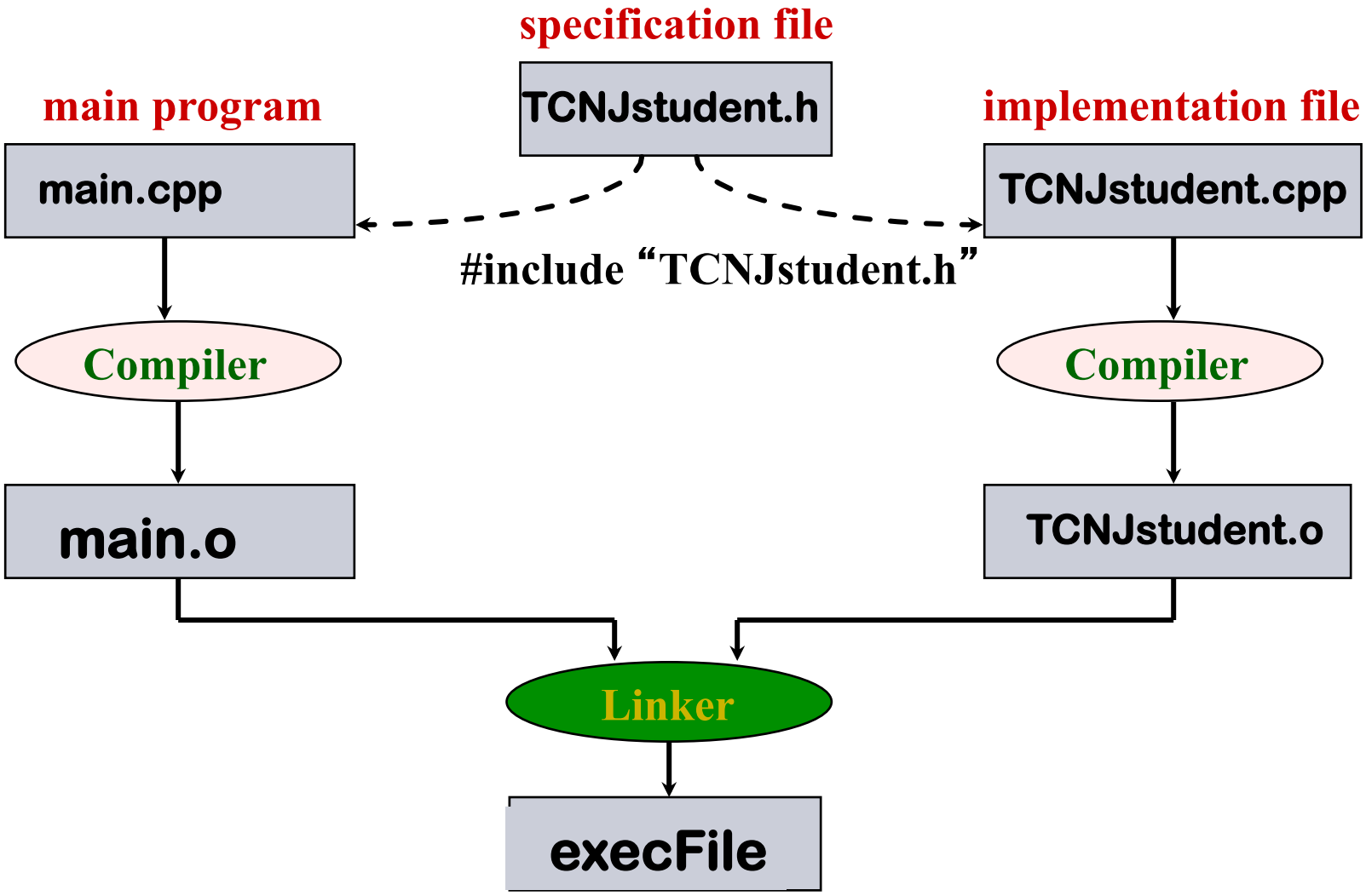
Must include the corresponding header file

Compile

g++ -o excuFile *main.cpp TCNJstudent.cpp*

Any executable filename you prefer

Separate Compilation and Linking of Files



Review: Parameter passing

Methods for passing parameters

- Passing by values
- Passing by references
- Passing by pointers

Outline

- Lab 2 discussion
- Dynamic memory allocation
- Object initialization
- Inheritance

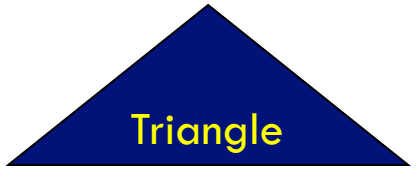
Inheritance

Polygon

```
class Polygon{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
};
```

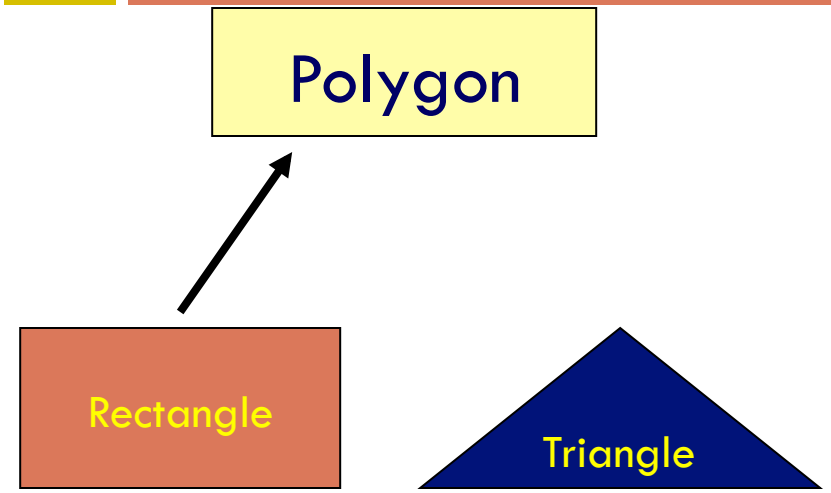
Rectangle

```
class Rectangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```




```
class Triangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

Inheritance

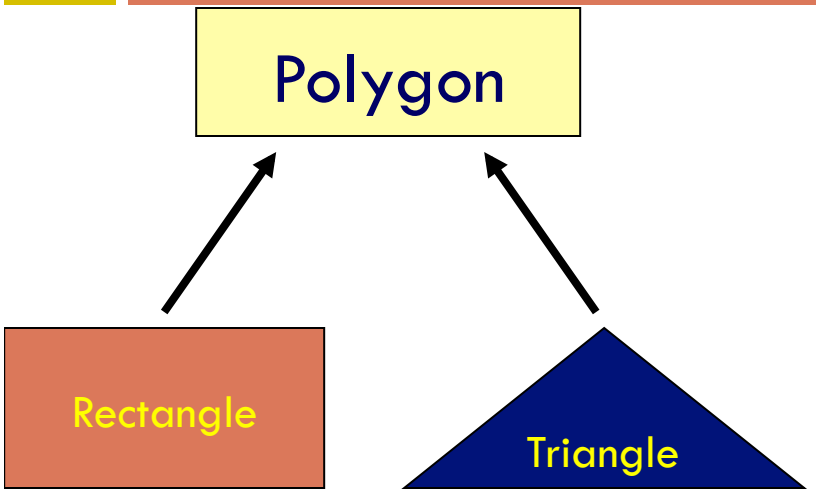



```
class Polygon{
  protected:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
};
```

```
class Rectangle : public Polygon{
  public:
    float area();
};
```

```
class Rectangle{
  protected:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

Inheritance




```
class Polygon{
  protected:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
};
```

```
class Triangle : public Polygon{
  public:
    float area();
};
```

```
class Triangle{
  protected:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

Base & Derived Classes

- Syntax:

  class *derived-class* : access-specifier *base-class*

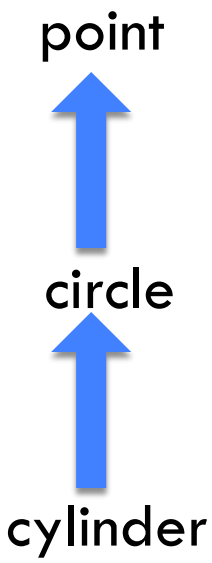
  where

  - access-specifier is one of **public, protected**, or **private**
    - private by default
    - Most of the time, people use public
- Any class can serve as a base class
  - Thus a derived class can also be a base class

Public & Inheritance

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

Class hierarchy




```
class Point{
  protected:
    int x, y;
  public:
    void set (int a, int b);
};
```

```
class circle : public point{
  private:
    double r;
  ... ...
};
```

```
 class cylinder : public circle{
    private:
      double h;
    ... ...
  };
```

Compare with Java

- Can you inherit from multiple classes in Java?

  class A extend class B, class C ?

- Can you do it in c++?
  - class C: public B, public A
  - **Examples—test_inhertitance.cpp**

What to inherit?

- In principle, every member of a base class is inherited by a derived class
  - just with different access permission