Classes: Advanced Topics

Jiun-Long Huang
National Chiao Tung University

Passing Objects to Functions

- Objects can be passed to functions in a similar way to the passing of any other value or variable.
- Both C and C++ compilers create copies of the values passed to functions in memory and destroy them when the functions are terminated.
- The question of whether or not the constructor and destructor functions are also invoked when creating and destroying the object's copies is raised.

Passing Objects to Functions (contd.)

- When passing an object to a function, its copy does not invoke a class constructor.
- When a function that has class objects as parameters terminates, the copies of the objects passed to the function are destroyed
  - The class destructor is called as many times as there are copies to destroy.

```
#include <iostream>
using namespace std;
int bcount=0; //counter that counts objects
class Box {
private:
  float side1, side2, side3;    //three sides of a box
public:
  Box(float s1, float s2, float s3) //constructor
  {
    side1=s1; side2=s2; side3=s3;   //Initializes sides
    bcount++;                       //Increments counter
  }
  ~Box()                            //destructor
  {
    bcount--;                 //Decrements counter
    cout<<"\t\tBox destroyed!\n";
  }
  float getVolume()                 //Returns volume
  {
    return  side1 * side2 * side3;
  }
};
```

```
float addBoxes(Box b1, Box b2)
//Returns the total volume of the two objects passed
{
  cout<<"\t\tAdding boxes:\n";
  cout<<"\t\t"<<bcount<<" boxes built so far.\n";
  return b1.getVolume()+b2.getVolume();
}

int main()
{
  Box a(1,2,3), b(2,3,4);
  cout<<bcount<<" boxes built so far.\n";
  cout<<"\t\tTotal volume = "<<addBoxes(a, b)<<endl;
  cout<<bcount<<" boxes built so far.\n";
  return 0;
}
```

Destructor is invoked two times

Constructor is invoked two times

Destructor is invoked two times

```
2 boxes built so far.
                Adding boxes:
                2 boxes built so far.
                Total volume = 30
                Box destroyed!
                Box destroyed!
0 boxes built so far.
                Box destroyed!
                Box destroyed!
```

Problem

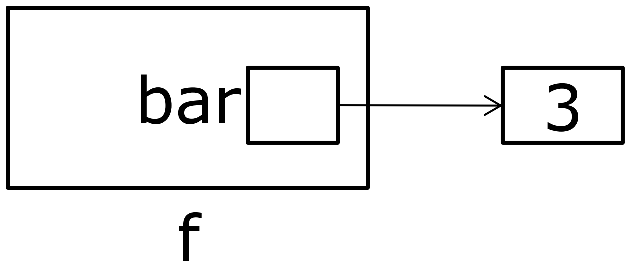
- The process of passing objects to functions may become a source of various logic errors
- Example:
  - Consider a class whose constructor allocates memory dynamically.
  - When the object's copy is destroyed, the destructor will free memory allocated for the original object, which will damage the original object.

Problem (contd.)

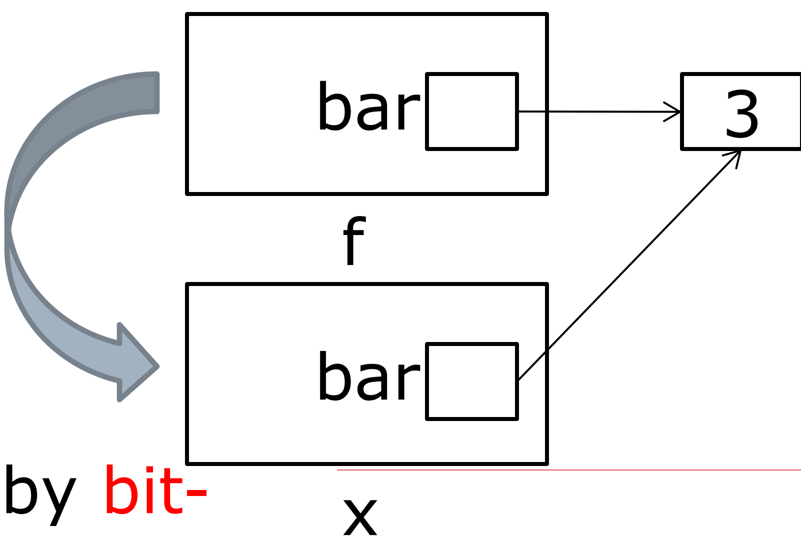
```
class Foo {
  int * bar;
public:
  Foo() {
    bar=new int(3);
  }
  ~Foo() {
    delete bar;
  }
};
```

```
Foo f;
void func(Foo x)
{
  return;
}
int main()
{
  func(f);
  return 0;
}
```

Problem (contd.)

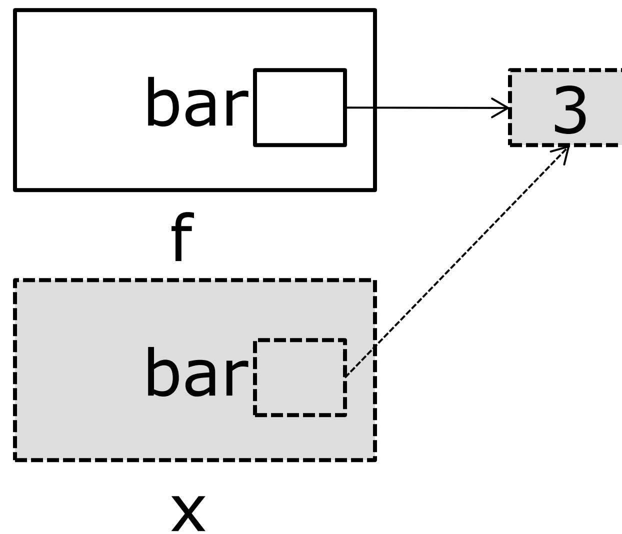


Invoke func(f)



Copy by bit-copy

func(f) terminates



Side Effect

- The total number of constructor and destructor calls in a program may not be matched, as the result of the operations performed when passing objects to functions.
- This mismatch may cause side effects, if the destructor function performs actions that should reverse the constructor's actions, such as freeing memory dynamically allocated by the constructor.

Copy Constructor

- The copy constructor is automatically called to initialize an object if one of the following three situations occurs
  - An object is used to initialize another object in a declaration statement
  - An object is passed to a function
  - An object is returned from a function
    - A temporary object may be created → invocation of copy constructor
    - Some compliers may perform some optimization techniques to eliminate some invocations of copy constructors

Example

```
Pixel p1;
  //Calls the default constructor to initialize p1
Pixel p2=p1 ;
  //p1 initializes p2; Calls the copy constructor (1)
Pixel p3(p2) ;
  //p2 initializes p3; Calls the copy constructor (1)
fun1(p1);
  //p1 is passed to fun1( );
  //Calls the copy constructor (2)
p2=fun2( ) ;
  //fun2( ) returns an object to p2;
  //Calls the copy constructor (3)
p3=p2;
  //The copy constructor is not called here.
  //NOTE: The copy constructor is not called when
  //assigning an object to another object.
```

Format

- A copy constructor has one parameter, which is a reference to the object to be copied.
- The const keyword precedes the reference because the original object should not be changed.

```
class_name (const class_name & object_name)
{
  //Body of the copy constructor
}
```

```cpp
#include <iostream>
using namespace std;
class Pixel{
   int x, y;
public:
   Pixel(int a, int b)                    //regular constructor
   {
       x=a; y=b;
       cout<<"\tNormal Constructor"<<endl;
   }
   Pixel(const Pixel &p)                  //copy constructor
   {
       x=p.x; y=p.y;
       cout<<"\tCopy Constructor"<<endl;
   }
   ~Pixel( ){ cout<<"\tDestructor"<<endl; }
   void setX(int x1) { x=x1; }
   void setY(int y1) { y=y1; }
   void showXY(){ cout<<"X="<<x<<" Y="<<y<<endl; }
};
```

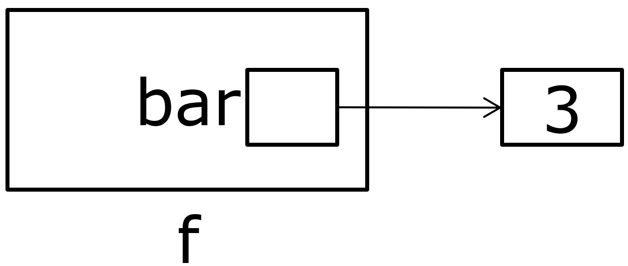
```
Pixel center(Pixel tp) //Sets x and y coordinates to 512
{                      //and returns the object
   tp.setX(512);
   tp.setY(512);
   return tp;
}

int main()
{
   Pixel p1(10, 20);   //Calls regular constructor
   p1.showXY();
   Pixel p2=p1;        //Calls the copy constructor
   p2.showXY();
   p2=center(p1);      //Calls the copy constructor twice
   p2.showXY();
   return 0;
}
```

```
          Normal Constructor
X=10 Y=20
          Copy Constructor
X=10 Y=20
          Copy Constructor
          Copy Constructor
          Destructor
          Destructor
X=512 Y=512
          Destructor
          Destructor
```

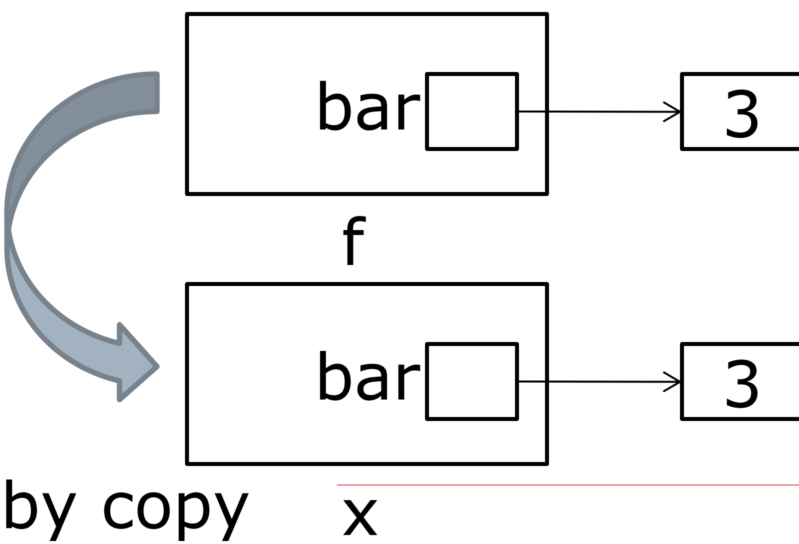
- The number of destructor calls is equal to the total number of all constructor calls (copy constructor calls plus regular constructor calls).
- If a class does not have an explicit copy constructor definition, the C++ compiler will create the default copy constructor, which will simply make an identical (bit-by-bit) copy of an object.
- Explicit copy constructor can solve the problem mentioned before

```
class Foo {
  int * bar;
public:
  Foo() {
    bar=new int(3);
  }
  Foo(const Foo & x) {
    bar=new int(x.bar);
  }
  ~Foo() {
    delete bar;
  }
};
```

```
Foo f;
void func(Foo x)
{
  return;
}
int main()
{
  func(f);
  return 0;
}
```

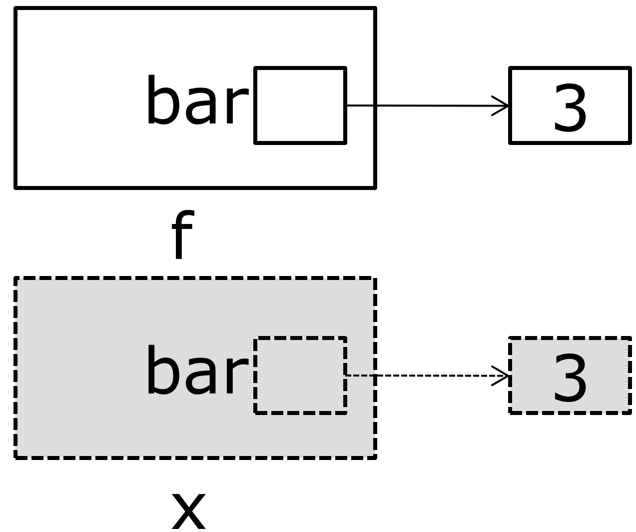


Invoke func(f)



Copy by copy constructor

func(f) terminates



Note on Copy Constructor

- Use copy constructor only when necessary
  - Performing copy constructor is time-consuming
- Overloading the assignment operator (=) when you implement a copy constructor
- Example:
  - If dynamic memory allocation is used in constructor
    - Implement the copy constructor, destructor, and overload the assignment operator

Friend Functions and Classes

- Hiding data inside a class and letting only class member functions have direct access to private data is a very important OOP concept.
- A friend function is not a member function but can still access class private members.
- A friend class is a class that can access class private members.

Friend Function

- There are several reasons for using friend functions and the most important are the following:
  - To have one function that can access private members of two or more different classes
  - To create some types of I/O functions
  - To design some types of operator functions (covered in Chapter 6)

Properties of Friend Function

- Properties:
  - Its prototype is placed inside the class definition and preceded by the friend keyword.
  - It is defined outside the class as a normal, non-member function.
  - It is called just like a normal non-member function.

```cpp
#include <iostream>
using namespace std;
class Count
{
  friend void setX( Count &, int );
public:
  Count() { x=0; }
  void print() const
  {
    cout << x << endl;
  }
private:
   int x;
};
void setX( Count &c, int val )
{
   c.x = val; // allowed because setX is a friend of Count
}
```

```
int main()
{
   Count counter; // create Count object
   cout << "counter.x after instantiation: ";
   counter.print();
   setX( counter, 8 );
   cout << "counter.x after call to setX: ";
   counter.print();
   return 0;
} // end main
```

```
counter.x after instantiation: 0
counter.x after call to : 8
```

Note

- A friend function cannot be inherited. Each parent or child class in the inheritance chain should have its own friends.
  - Inheritance issues will be discussed in Chapter 7.
- A friend function may be a member of one class and a friend of another

Friend Classes

- An entire class can be a friend of another class.
  - All member functions of one class should access the data of another class.
- To make the A class a friend of the B class, the friend class keyword must precede the A class name and be placed within the body of the B class.

- The A class should be defined prior to the B class or its forward reference must be placed before the B class definition as follows:

```
class A
{
   ...
};
class B
{
   ...
   friend class A;
};
```

```
class B;
class A
{
   ...
   friend class B;
};
class B
{
   ...
   friend class A;
};
```

Friendship is NOT

- Reverse
  - If A is a friend of B, it does not mean that B is a friend of A, unless specified explicitly.
- Transitive
  - If A is a friend of B and B is a friend of C, it does not mean that A is a friend of C.
- Inherited
  - If A is a friend of B, it does not mean that A is a friend of a child of B as well (Chapter 7).

The this Pointer

- The this pointer stores the address of an object used to call a non-static member function.
- Most of the time the this pointer is hidden from programmers and is handled and processed implicitly by the compiler.
- Programmers can use the pointer explicitly as well.

Example

```
class Pixel {
  int x, y;
public:
  //the this pointer is hidden here
  void set(int a, int b) ( x=a; y=b; }
  void get() { cout<<x<<y; }
}
```

```
class Pixel {
  int x, y;
public:     //the this pointer is used explicitly
  void set(int a, int b) { this->x=a; this->y=b; }
  void get() {cout << this->x<<this->y; }
}
```

```
#include <iostream>
using namespace std;
class Pixel{
  int x, y;
public:
  Pixel() { cout<<"-> Pixel created!"<<endl; x=0; y=0; }
  ~Pixel() { cout<<"-> Pixel destroyed!"<<endl; }
  void setCoord(int x1, int y1) { x=x1; y=y1; }
  //void setCoord(int x, int y) { this->x=x; this->y=y; }
  void getCoord()
  {
    cout<<"Pixel's coordinates:"<<endl;
    cout<<"X="<<x<<" Y="<<y<<endl;
  }
  Pixel move_10()
  {
    x = x + 10;
    y = y + 10;
    return *this;
  }
};
```

```cpp
int main()
{
  Pixel p1, p2;
  int x1, y1;
  cout<<"Enter X and Y coordinates =>";
  cin>>x1>>y1;
  p1.setCoord(x1,y1);
  p1.getCoord();
  p2 = p1.move_10();
  p2.getCoord();
  p1.getCoord();
  return 0;
}
```

```
-> Pixel created!
-> Pixel created!
Enter X and Y coordinates
=>5 6
Pixel's coordinates:
X=5 Y=6
-> Pixel destroyed!
Pixel's coordinates:
X=15 Y=16
Pixel's coordinates:
X=15 Y=16
-> Pixel destroyed!
-> Pixel destroyed!
```

Static Members

- Static data members belong to the class itself.
- C++ creates one copy of each static data member in memory, no matter how many class objects are instantiated.
- All objects of the class therefore share the same copy of a static member.
- To make a data member static, the static keyword must precede the data member name when it is declared inside the class.

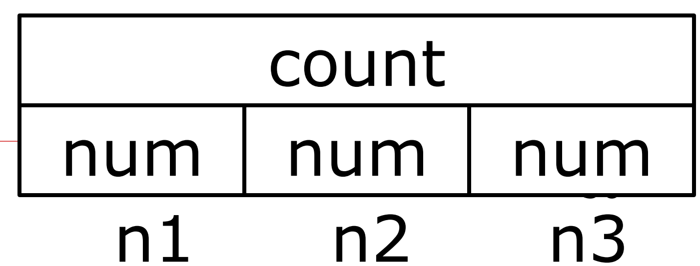
Static Data Member

- All static data members exist in memory before any object of their class is instantiated.
- Being independent from objects, they are good candidates for the following:
  - Counters that count the number of objects instantiated or destroyed
  - Totals that accumulate values stored in objects
  - Constants

```
#include <iostream>
using namespace std;
class Node {
public:
  static int count;      Declaration
  int num;
  Node(){ num=1; count++; }
  ~Node(){ count--; }
  void print(void);
  void add(void) {num++;}
};
void Node::print()
{
    cout<<"count= "<<count<<" num= "<<num<<endl;
}
//Allocates memory for static data members
int Node::count=0;      Initialization
```

```cpp
int main()
{
  cout<<"\ncount= "<<Node::count<<endl;
  //cout<<"\nnum= "<<Node::num<<endl;     //ERROR
  Node *n1, *n2, *n3;
  n1=new Node();
  n2=new Node();
  n3=new Node();
  n1->add();
  n1->print();
  n2->print();
  n3->print();
  cout<<"\ncount "<<Node::count<<endl;
  delete n1;
  cout<<"\ncount "<<Node::count<<endl;
  return 0;
}
```

```
count= 0
count= 3 num= 2
count= 3 num= 1
count= 3 num= 1

count 3

count 2
```



Static Member Function

- A member function can be declared static simply by preceding the function return type with the static keyword in a class definition.

Differences between Static and Non-static Member Functions

- A static member function is not attached to any object.
  - An object is not needed when calling static member functions.
- A static member function does not have direct access to the private class data members.
- A static member function does not have a this pointer (the this pointer will be discussed in the next section).

```cpp
#include <iostream>
using namespace std;
class Node {
private:
 static int count;
public:
  int num;
  Node(){ num=1; count++; }
  ~Node(){ count--; }
  void print(void);
  void add(void) {num++;}
  static int getCount(void) { return count; }
};
void Node::print()
{
    cout<<"count= "<<count<<" num= "<<num<<endl;
}
//Allocates memory for static data members
int Node::count=0;
```

```
int main()
{
  cout<<"\ncount= "<<Node::getCount()<<endl;
  Node *n1, *n2, *n3;
  n1=new Node();
  n2=new Node();
  n3=new Node();
  n1->add();
  n1->print();
  n2->print();
  n3->print();
  cout<<"\ncount "<<Node::getCount()<<endl;
  delete n1;
  cout<<"\ncount "<<Node::getCount()<<endl;
  return 0;
}
```

Class Math in Java-Style

```
class Math
{
public:
  static double abs(double a);
  ...
  static double pow(double a, double b);
  ...
};
----------------------------
cout<<Math::abs(-1.5);
```

Using const Member Functions

- Unlike regular non-const member functions, a const member function cannot modify the object that is used to invoke the function.
- To declare a member function as const, the const keyword must be inserted after the closing bracket of the parameter list, in both the function prototype and function definition.

```
//const member function prototype
 return-type function_name(parameter list) const;

//coast member function definition
return_type class_name::function_name(parameter
list) const
{
//body of the function
}
```

```
class Power
{
private:
  float voltage;
  float frequency;
public:
  Power(float v, float f) {voltage=v; frequency=f;}
  void display()
    {cout<<voltage<<" "<<frequency<<endl;}
  float getVolt() {return voltage;}
};
```

```
const Power source (110, 60);
Power bat(12,0);
float v1, v2;
// ERROR: non-const function cannot process
// const object
v1=source.getVolt();
v2=bat.getVolt();
// ERROR: non-const function cannot process
// const object
source.display();
bat.display();
```

```
class Power
{
private:
  float voltage;
  float frequency;
public:
  Power(float v, float f) {voltage=v; frequency=f;}
  void display const ()
    {cout<<voltage<<" "<<frequency<<endl;}
  float getVolt() const {return voltage;}
};
```

Note

- When using const member functions and const objects, each of the following actions will cause a syntax error:
  - Attempting to modify class data members
  - Declaring a constructor or destructor as const
  - Attempting to modify a const object by using the assignment operator
  - Calling a non-const member function from the body of a const function

```
//ERROR; this function cannot be const
void set(float v, float f ) const {

  voltage = v;
  frequency = f;
}
//ERROR; this function cannot be const
void setVolt() const
{
  cout<<"Enter voltage:";
  cin>>voltage;
}
```