Copy Ctor

Object-Oriented Programming with C++

Zhaopeng Cui

Copying

- Create a new object from an existing one
  - For example, when calling a function

```
// Currency as pass-by-value argument
void func(Currency p) {
  cout << "X = " << p.dollars();
}
...
Currency bucks(100, 0);
func(bucks); // bucks is copied into p
```

Example: HowMany.cpp

The copy constructor

- Copying is implemented by the ***copy constructor***
- Has the unique signature

```
T::T(const T&);
```

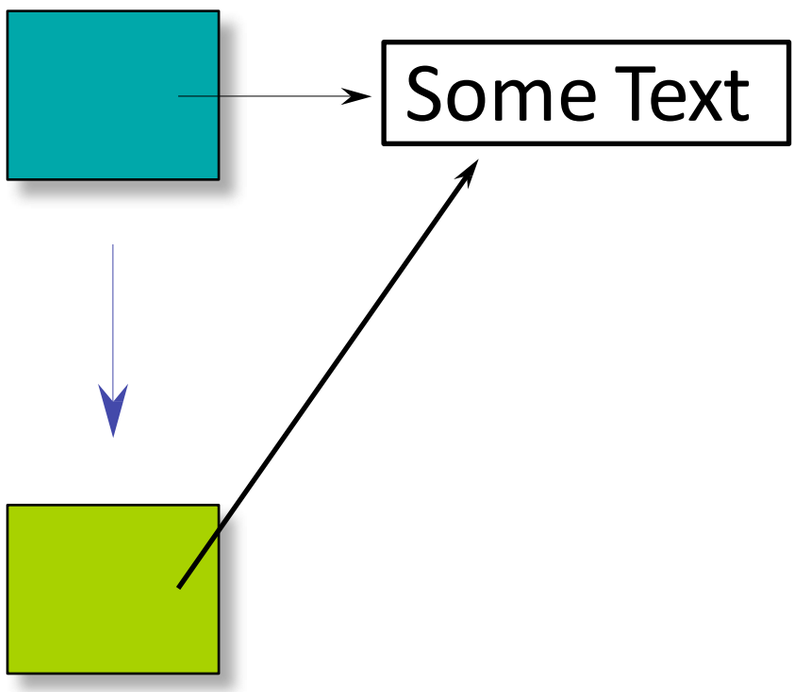
  – Call-by-reference is used for the explicit argument
- C++ builds a copy ctor for you if you don't provide one!
  – Copies each member variable
    - Good for numbers, objects, object arrays
  – Copies each pointer
    - Data may become shared!

What if class contains pointers?

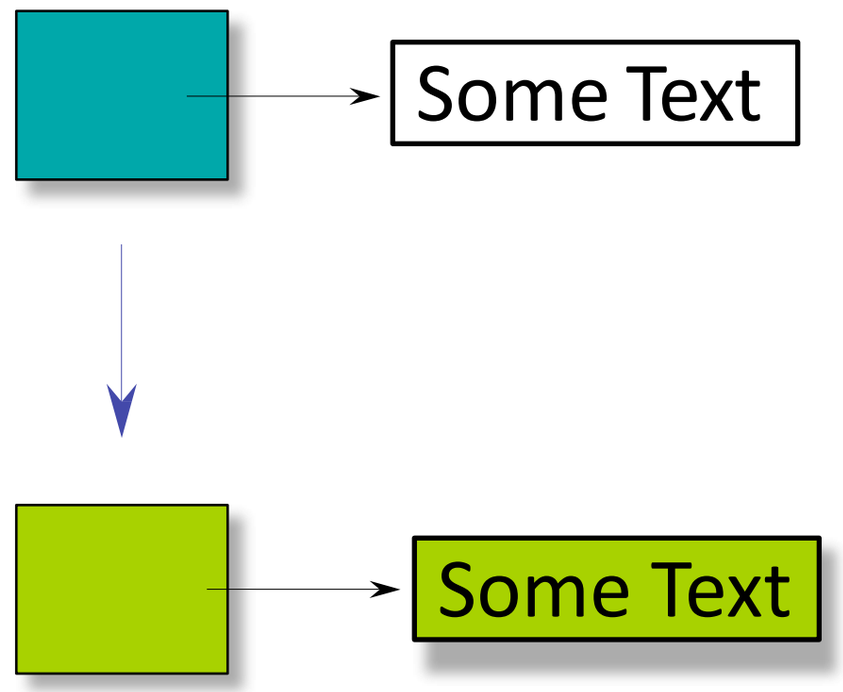
```
class Person {
public:
   Person(const char *s);
   ~Person();
   void print();
   // ... accessor functions
private:
   char *name;   // char * instead of string
   //... more info e.g. age, address, phone
};
```

Choices

Copy pointer



Copy entire block

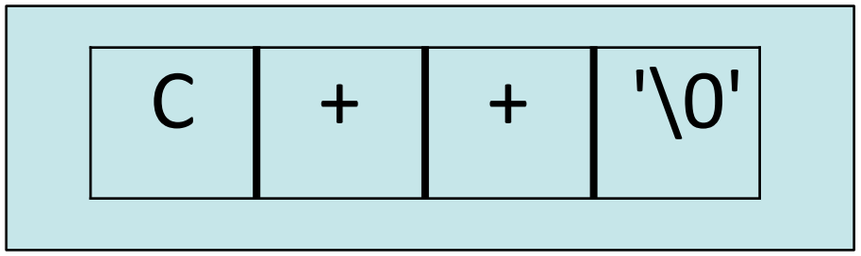


code & demo

Character strings

- In C++, a character string is
  - An array of characters
  - With a special terminator — ‘\0’ or ASCII null

- The string “C++” is represented, in memory, by an array of four (4, count’em) characters



Standard C library String fxns

- Declared in `<cstring>`

  `size_t strlen(const char *s);`

  – s is a null-terminated string

  – returns the length of s

  – length does not include the terminator!

  `char *strcpy (char *dest, const char *src);`

  – Copies src to dest stopping after the terminating null-character is copied. (src should be null-terminated!)

  – dest should have enough memory space allocated to contain src string.

  – Return Value: returns dest

Person (char*) implementation

```
#include <cstring>      // #include <string.h>
using namespace std;

Person::Person( const char *s ) {
 name = new char[strlen(s) + 1];
  strcpy(name, s);
}

Person::~Person() {
  delete [] name;      // array delete
}
```

Person copy constructor

- To Person declaration add copy ctor prototype:

```
Person( const Person& w );  // copy ctor
```

- To Person .cpp add copy ctor defintion:

```
Person::Person( const Person& w ) {
    name = new char[strlen(w.name) + 1];
    strcpy(name, w.name);
}
```

- No value returned
- Accesses **w.name** across client boundary
- The copy ctor initializes uninitialized memory

Person: string name

- What if the name was a string (and not a char*)

```
#include <string>
class Person {
public:
    Person( const string& );
    ~Person();
    void print();
    // ... other accessor fxns ...
private:
    string name;    // embedded object (composition)
    // ... other data members...
};
```

Person: string name…

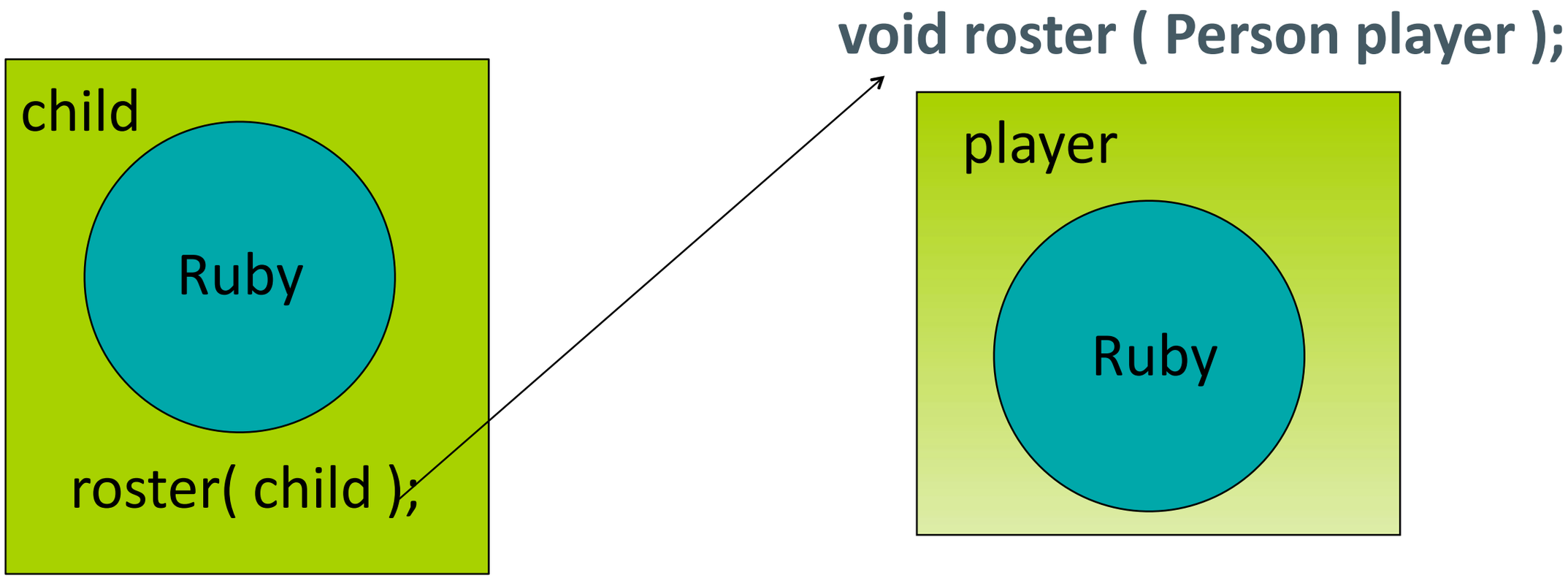
- In the default copy ctor, the compiler recursively calls the copy ctors for all member objects (and base classes).

- default is *memberwise* initialization

When are copy ctors called?

- During call by value

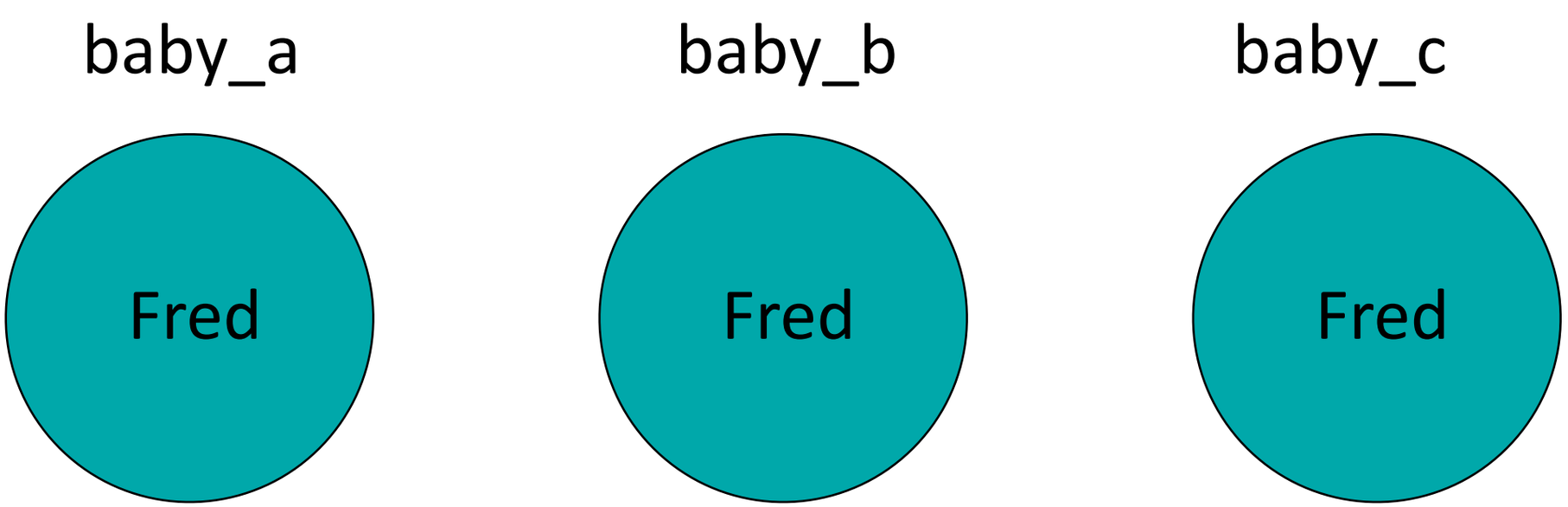
```
void roster( Person );        // declare function
Person child( "Ruby" );       // create object
roster( child );    // call function
```



When are copy ctors called?

- During initialization

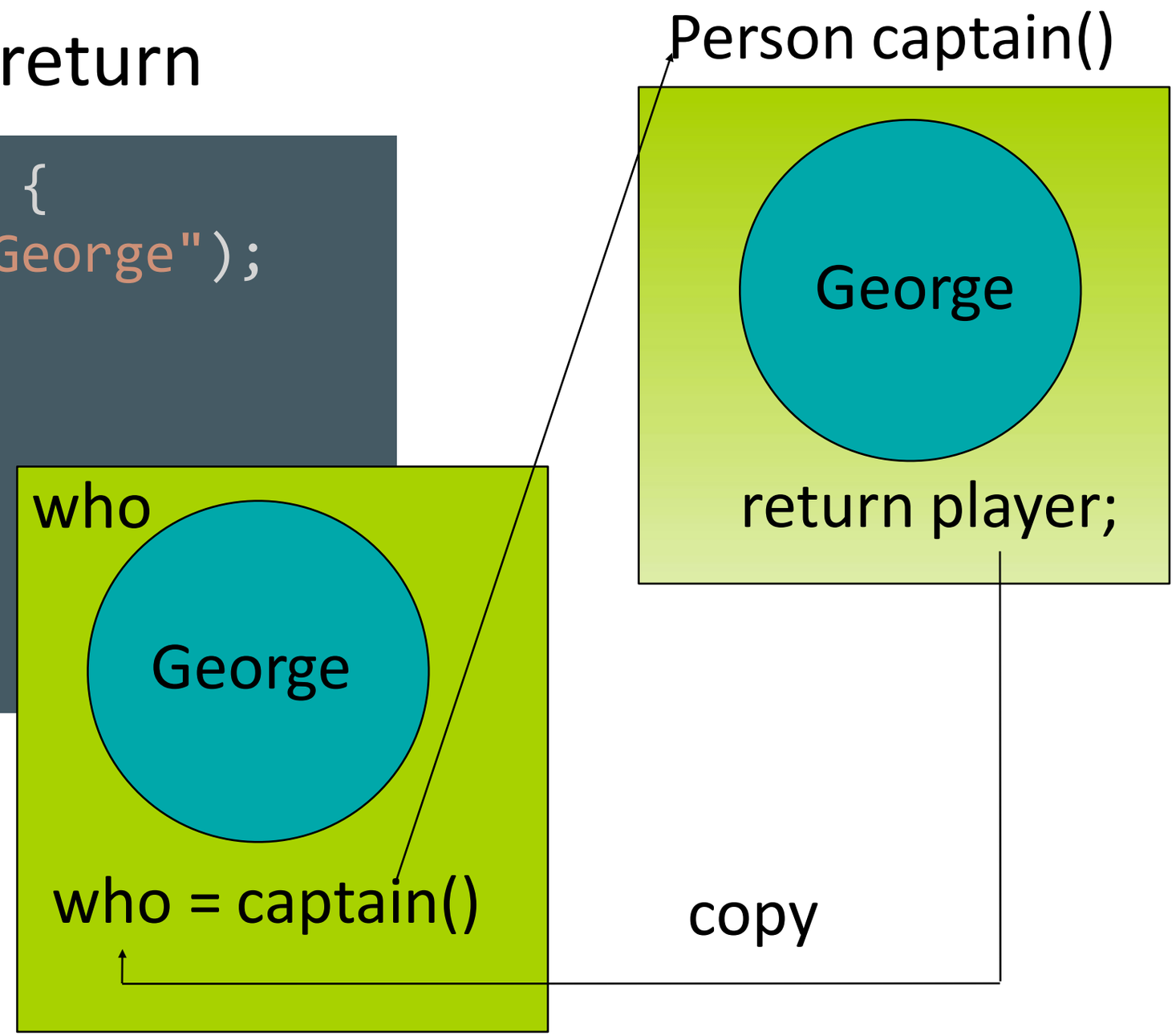
```
Person baby_a("Fred");

// The followings use the copy ctor
Person baby_b = baby_a;    // not an assignment
Person baby_c( baby_a );   // not an assignment
```



When are copy ctors called?

- During function return

```
Person captain()    {
    Person player("George");
    return player;
    }
    ...
    Person who
        = captain();
    ...
```



Copies and overhead

- Compilers can "optimize out" copies when safe!

- Programmers need to
  - Program for “dumb” compilers
  - Be ready to look for optimizations

Example

```
Person copy_func( Person p ) {
    p.print();
    return p;
} // copy ctor called!

Person nocopy_func(const char *who ) {
    return Person( who );
}  // no copy needed!
```

code & demo

Constructions vs. assignment

- Every object is constructed once

- Every object should be destroyed once
  - Forget to invoke `delete`
  - Invoke `delete` more than once

- Once an object is constructed, it can be the target of many assignment operations

Copy ctor guidelines

- In most cases, you don’t have to write.

- Be explicit when necessary, e.g., managing raw pointers.
  - create your own copy ctor

- If you don't need one, declare a private copy ctor (no need to define the body).
  - prevents creation of a default copy constructor
  - generates a compiler error for copy
  - use the following (since C++11):

```
Person(const Person &rhs) = delete;
```

static

Static in C++

- Two basic meanings:
  - Static storage
    - allocated once at a fixed address
  - Visibility of a name
    - internal linkage

Uses of “static” in C++

Where to use	What does it mean
Static free functions	Internal linkage
Static global variables	Internal linkage
Static local variables	Persistent storage
Static member variables	Shared by all instances
Static member function	Shared by all instances, can only access static member variables

Global static hidden in file

. cpp file 1

```
int g_global;
static int s_local;

void  func()
{
...
}

static  void
hidden()
{
...
}
```

.cpp file 2

```
extern int g_global;
void func();

extern int s_local;
int myfunc() {

  g_global += 2;
  s_local *= g_global;
  func();
}
```

?

Static inside functions

- Value is remembered for entire program
- Initialization occurs only once
- Example:
  - count the number of times the function has been called

```
void f() {
    static int num_calls = 0;
  ...
  num_calls++;
}
```

Static applied to objects

- Suppose you have a class

```
class X {
    X(int, int);
    ~X();
    ...
};
```

- And a function with a static X object

```
void f() {
    static X my_X(10, 20);
    ...
}
```

Static applied to objects…

- Construction occurs when definition is encountered
  - Constructor called at-most once
  - The constructor arguments must be satisfied

- Destruction takes place on exit from *program*
  - Compiler assures LIFO order of destructors

Conditional construction

- Example: conditional construction

```
void f(int x) {
    if (x > 10) {
        static X my_X(x, x * 21);
        ...
    }
}
```

- **my_X**
  - is constructed once, if f() is ever called with x >10
  - retains its value
  - destroyed only if constructed

Global objects

- Consider

```
#include "X.h"
static X global_x(12, 34);
static X global_x2(8, 16);
```

- Constructors are called before main() is entered
  - Order controlled by appearance in file
  - In this case, `global_x` before `global_x2`
  - **main()** is "no longer" the *first* function being called

- Destructors called when
  - **main()** exits
  - **exit()** is called

Can we apply static to members?

- Static means
  - Hidden
  - Persistent
- Hidden: *A static member is a member*
  - Obeys usual access rules
- Persistent: *Independent of instances*
- Static members are class-wide
  - variables or
  - functions

Static members

- Static member variables
  - –Global to all class member functions
  - –Initialized once, at file scope
  - –Provide a place for this variable and init it in .cpp
  - –No ‘static’ when it is initialized in .cpp

- Example: StatMem.cpp

Static members

- Static member functions
  - –Have no implicit receiver ("this")
    - (why?)
  - –Can access only static member variables
    - (or other globals)
  - –Can’t be dynamically overridden

- Example: StatFun.cpp

To use static members

- <class name>::<static member>
- <object variable>.<static member>