Copy Constructor

Object-Oriented Programming with C++

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
```

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

The copy constructor

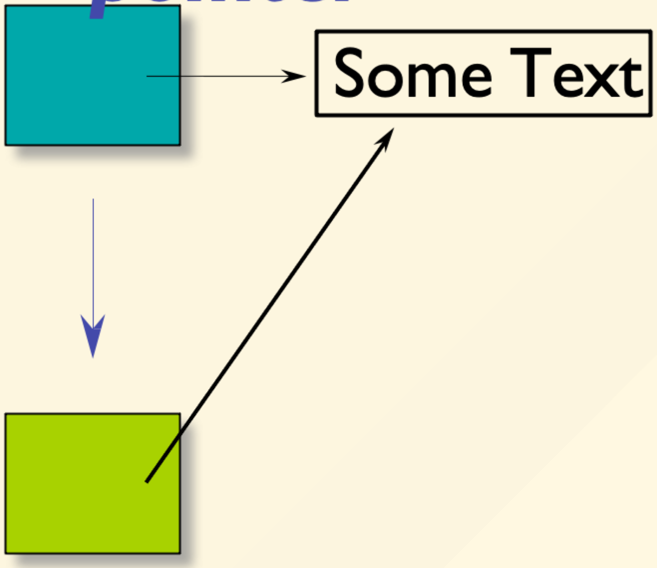
- C++ builds a copy ctor for you if you don't provide one!
    - Copies each member variable
        - Good for numbers, objects, object arrays
    - Copies each pointer
        - Data may become shared!

What if a class contains pointers?

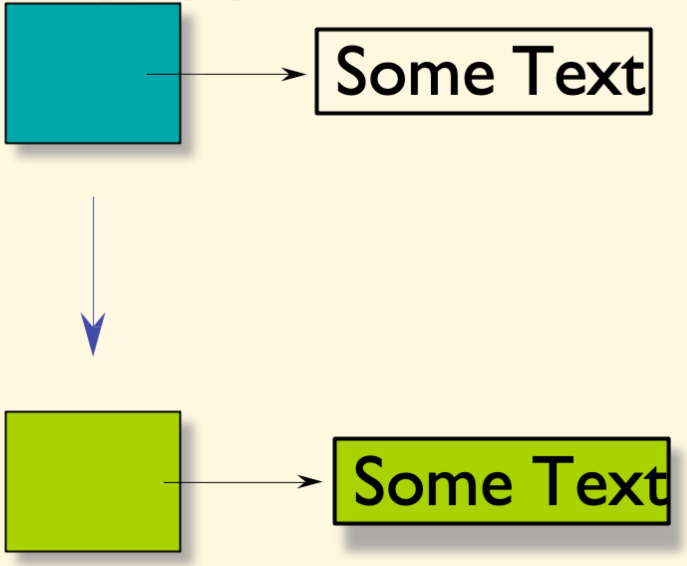
```
class Person {
public:
  Person(const char *s);
  ~Person();
  void print();
  // ... accessor functions
private:
  char *name;   // char * instead of string
  //... more info e.g. age, address, phone
};
```

Choices

Copy pointer



Copy entire block



Person (char*) implementation

```cpp
#include <cstring>
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.h* add copy ctor prototype:

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

Person copy constructor

- 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
- Allocates memory and initializes variable

Person (string) implementation

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

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

Person (string) implementation

- Default copy ctor: *memberwise* initialization
  - Recursively calls the copy ctors for all member objects (and base classes).

When are copy ctors called?

- During initialization

```
Person baby_a("Fred");

Person baby_b = baby_a;   // not an assignment
Person baby_c( baby_a );  // not an assignment
```

When are copy ctors called?

- During call by value

```
void roster( Person ) {
  ...
}

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

When are copy ctors called?

- During function return

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

Pay attention to efficiency

- Example: using the `vector<>` container
  - Estimate and preserve the memory
  - Avoid extra copies

Constructions vs. assignment

- Every object is constructed once
- Every object should be destroyed 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 anything.
- Be explicit when necessary, e.g., managing raw pointers.
  - create your own copy ctor

Copy ctor guidelines

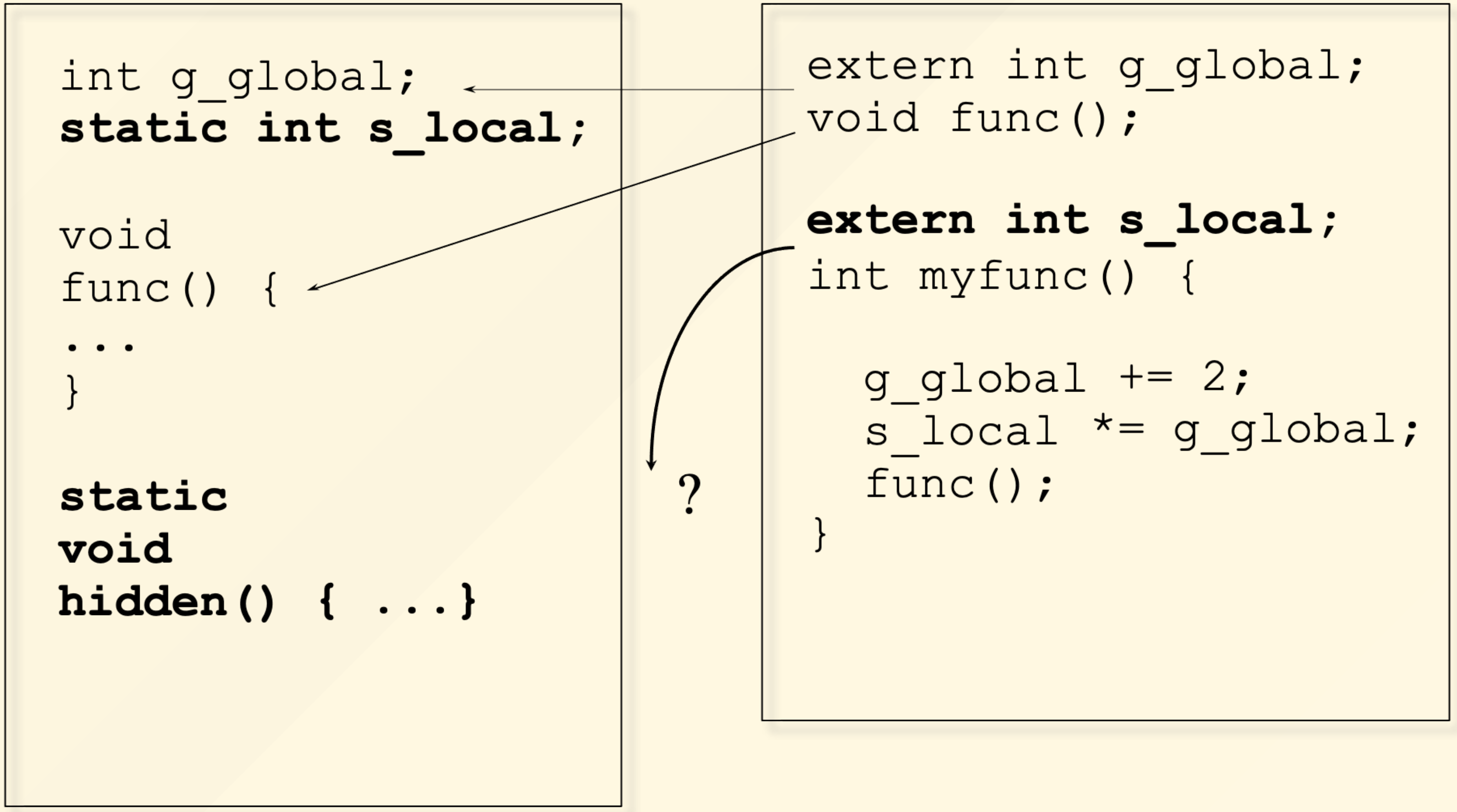
- If you want to forbid copy, then declare (no need to define the body) the copy ctor *private*.
    - prevents creation of a default copy constructor
    - generates a compiler error on copy behavior
    - use `Person(const Person &rhs) = delete;` (since C++11)

static

Static in C++

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

Global static stay 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();
}
```

Uses of `static` in C++

where to use	what does it mean
free functions	Internal linkage
global variables	Internal linkage
local variables	Persistent storage
member variables	Shared by all instances
member functions	Access static members only

Static applied to objects

- Suppose you have a class X

```
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 of *whole program*
    - Compiler assures LIFO order of destructors

Conditional construction

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

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

Global objects

```
#include "X.h"
static X global_x1(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_x1` before `global_x2`
  - `main()` is *no longer* the first function being called

Global objects

```
#include "X.h"
static X global_x1(12, 34);
static X global_x2(8, 16);
```

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

Can we apply static to members?

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

Static members

- Static member *variables*
  - Global to all class member functions
  - Defined and initialized in .cpp file
    - Do not write the `static` keyword

Static members

- Static member *functions*
  - Have no implicit receiver (`this`)
  - Can only access static members
  - Can’t be dynamically overridden

To use static members

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