CS100
Introduction to Programming

Lecture 16. Copy Constructor

Today's learning objectives

- Understand the Copy Constructor
- Static in C++
- Strings and Basic I/O

Outline

- Understand the Copy Constructor
- Static in C++
- Strings and Basic I/O

References as class members

- Declared without initial value.
- Must be initialized using constructor initializer list

```
class X {
public:
    int& m_y;
    X(int& a);
};
X::X(int& a) : m_y(a) { }
```

Returning references

- Functions can return references
  - But they should refer to non-local variables!

```
#include <assert.h>
const int SIZE = 32;
double myarray[SIZE];
double& subscript(const int i) {
    return myarray[i];
}

int main(void) {
    for (int i = 0; i < SIZE; i++) {
        myarray[i] = i * 0.5;
    }
    double value = subscript(12);
    subscript(3) = 34.5;
}
```

Returning references

- Functions can return references
  - But they should refer to non-local variables!

```
#include <assert.h>
const int SIZE = 32;
double myarray[SIZE];
double& subscript(const int i) {
    return myarray[i];
}

int main(void) {
    for (int i = 0; i < SIZE; i++) {
        myarray[i] = i * 0.5;
    }
    double value = subscript(12);
    subscript(3) = 34.5;
}
```

const in Functions Arguments

- Pass by const value -- don't do it
- Passing by const reference

```
Person(const string& name, int weight);
```

  - don't change the string object
  - more efficient to pass by reference (address) than to pass by value (copy)
  - const qualifier protects from change

```
// y is a constant! Can't be modified
void func(const int& y, int& z) {
    z = z * 5; // ok
    y += 8; // error!
}
```

Temporary values are const

- What you type

```
void func(int &);
func(i * 3); // Generates warning or error!
```

- What the compiler generates

```
void func(int &);
const int tmp@ = i * 3;
func(tmp@); // Problem -- binding const ref to
            // non-const argument!
```

The temporary is constant, since you can't access it

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
- C++ builds a copy ctor if you don't provide one!
  - Copies each member variable:

    Good for numbers, objects, arrays
  - Copies each pointer

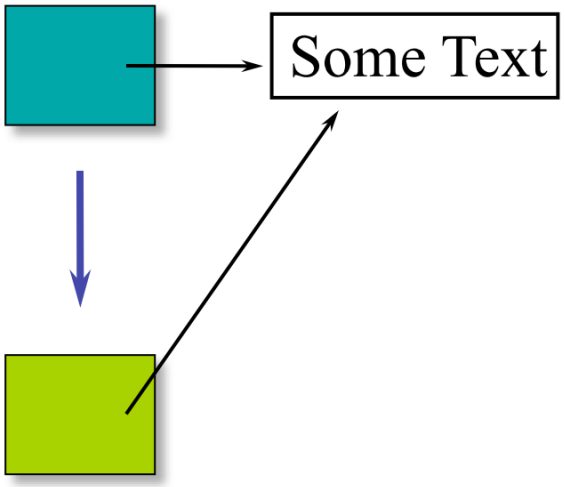
    Data may become shared!

What if class contains pointers?

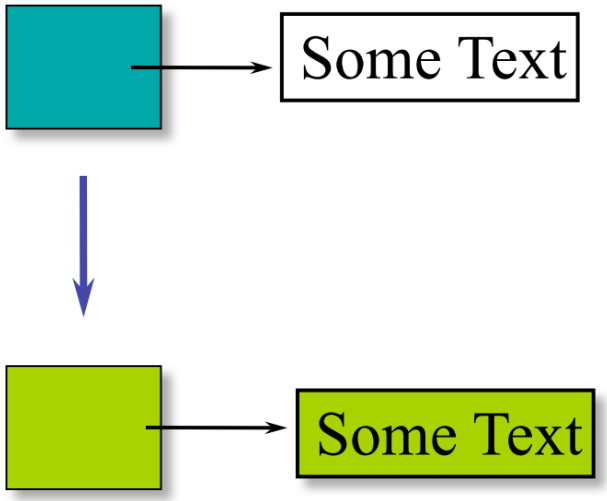
```
class Person {
    public:
    Person(const char *s);
    ~Person();
    void print(); // ... accessor functions
private:
    char *name;   // char * instead of string
    //... more info e.g. age, address, phone
};
```

Choices

Copy pointer

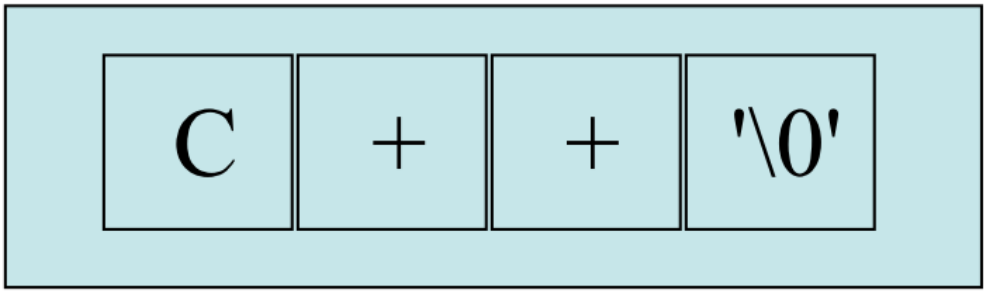


Copy entire block



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 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.

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:

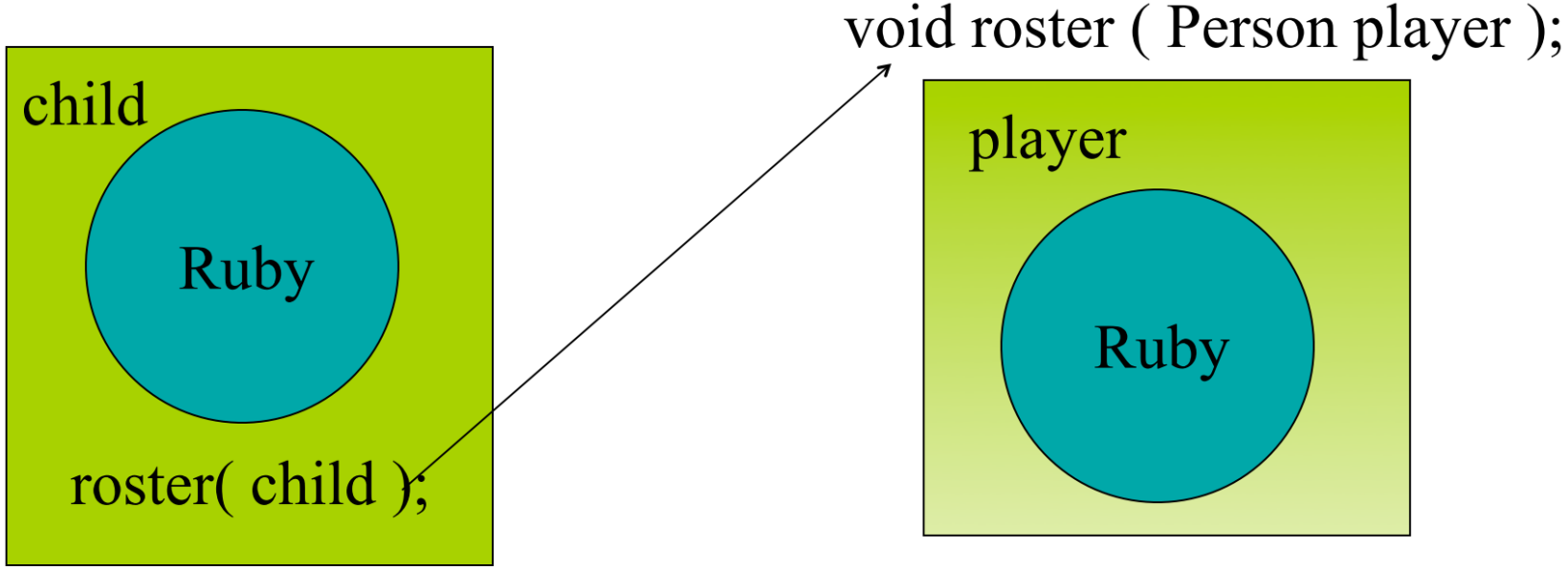
```
Person::Person(const Person& w) {
    name = new char[strlen(w.name) + 1];
    strcpy(name, w.name);
}
```

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

When are copy ctors called?

- During call by value

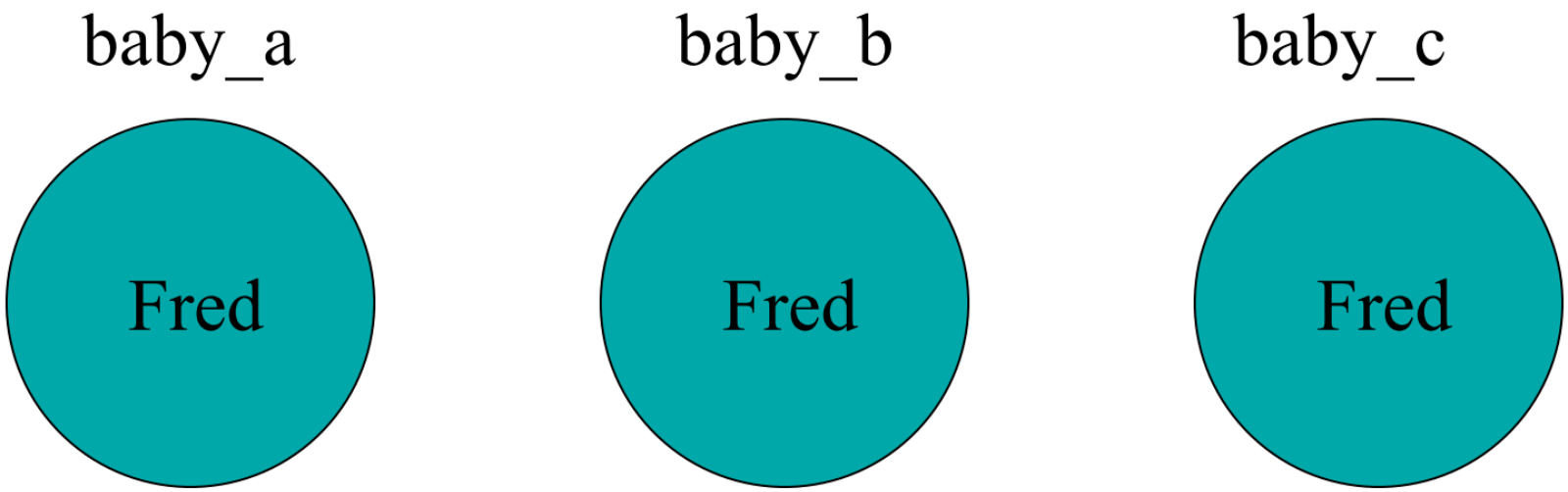
```
void roster(Person);          // declare function
Person child("Ruby");         // create object
roster(child);                // call function
```



When are copy ctors called?

- During initialization

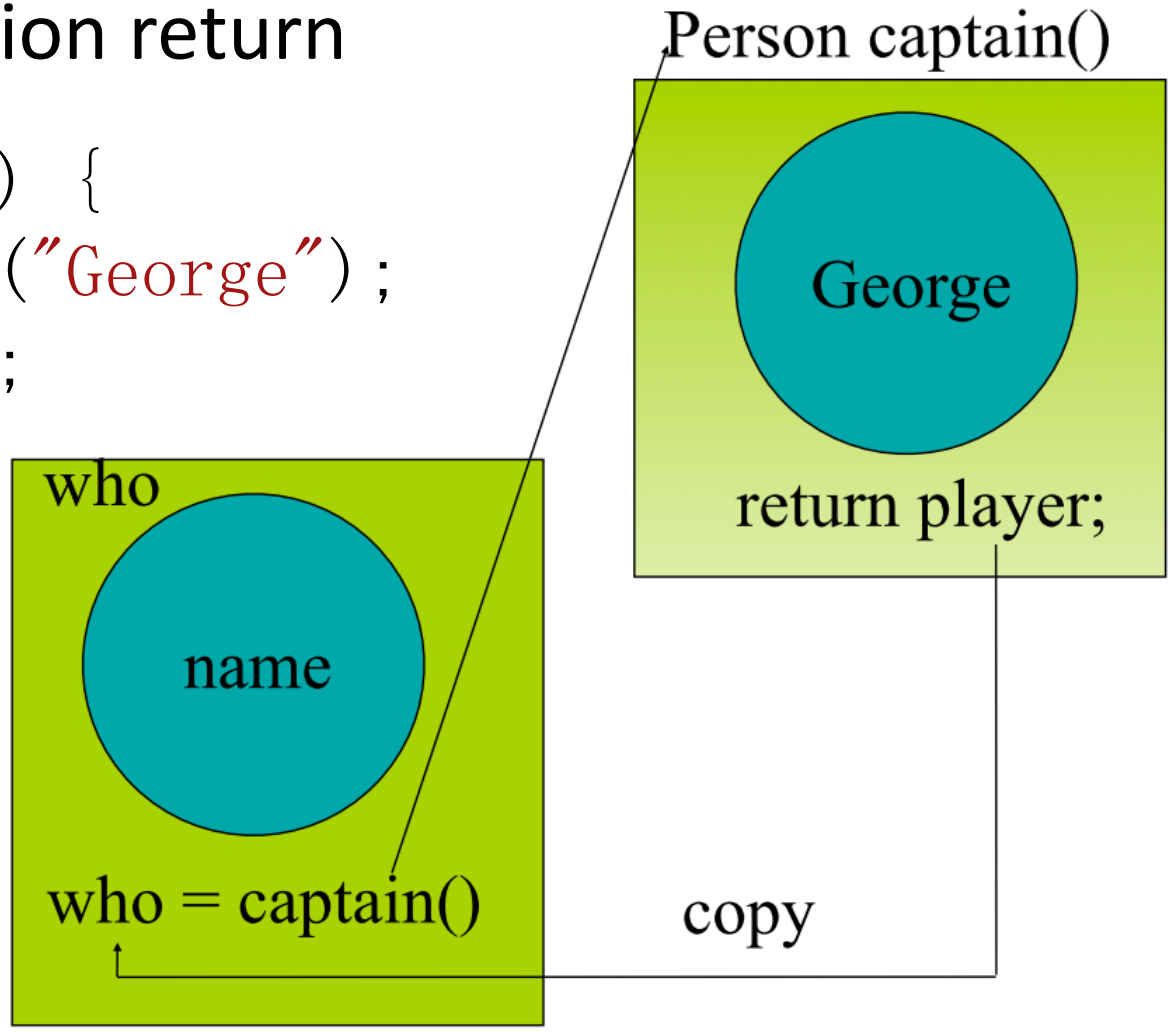
```
Person baby_a("Fred");
// these use the copy ctor
Person baby_b = baby_a;    // not an assignment
Person baby_c(baby_a);     // not an assignment
```



When are copy ctors called?

- During function return

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



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(char *who) {
    Person local(who);
    local.print();
    return local; // copy ctor called!
}

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

Constructions vs. assignment

- Every object is constructed once
- Every object should be destroyed once
  - Failure to invoke delete()
  - Invoking delete() more than once
- Once an object is constructed, it can be the target of many assignment operations

Person: string name

- What if the name was a string (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 ctor, the compiler recursively calls the copy ctors for all member objects (and base classes).

- Default is member-wise initialization

Copy ctor guidelines

- In general, be explicit
  - Create your own copy ctor
  - Don't rely on the default
- If you don't need one, declare a private copy ctor
  - Prevents creation of a default copy constructor
  - Generates a compiler error if try to pass-by-value
  - Don't need a definition

Outline

- Understand the Copy Constructor
- Static in C++
- Strings and Basic I/O

Static in C++

Two basic meanings

- Static storage

  - allocated once at a fixed address

- Visibility of a name

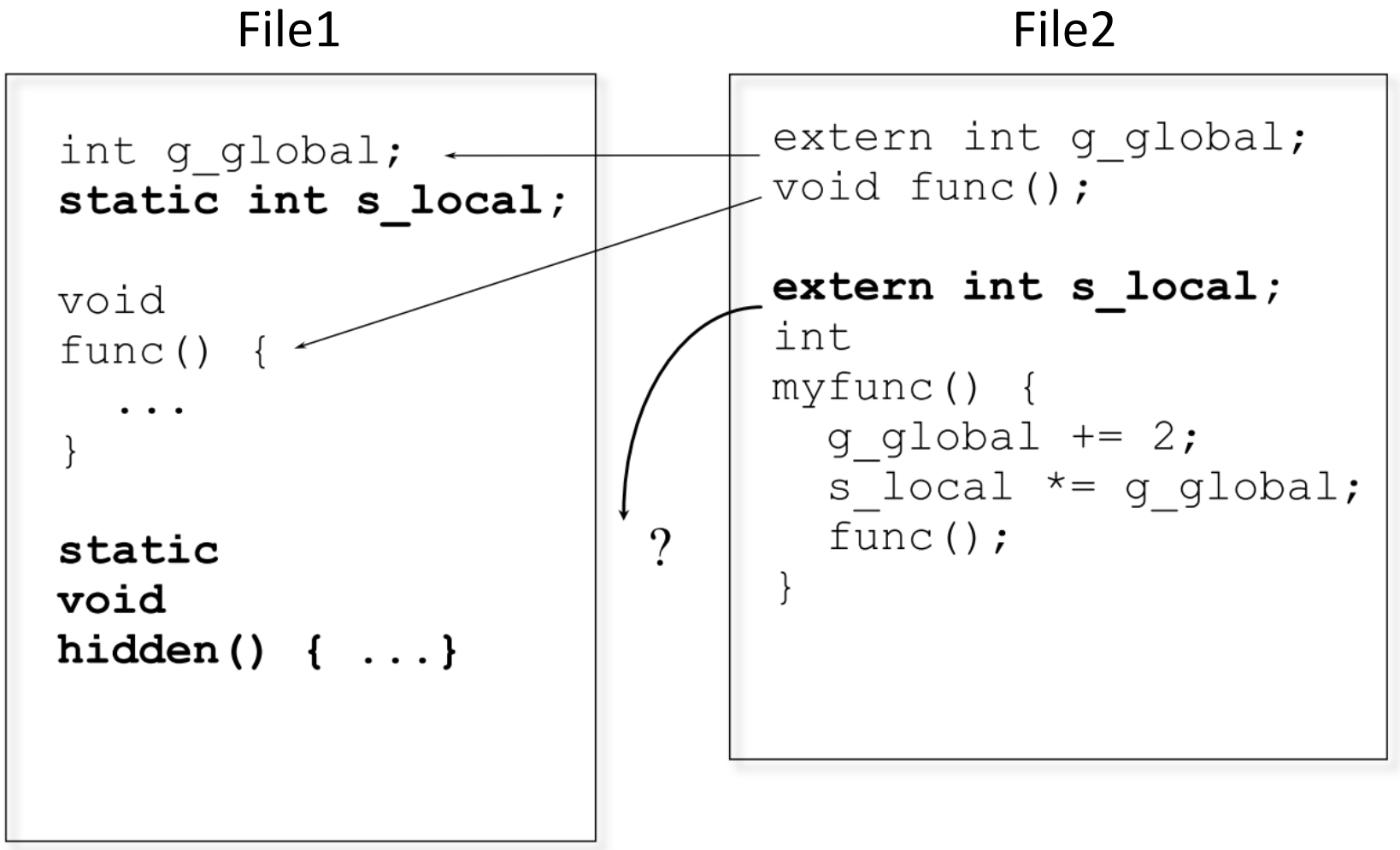
  - internal linkage

- Don't use static except inside functions and classes.

Uses of “static” in C++

Static free functions	~~Internal linkage~~ *(deprecated)*
Static global variables	~~Internal linkage~~ *(deprecated)*
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



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"
X global_x(12, 34);
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 called
- Destructors called when
  - main() exits
  - exit() is called

Static Initialization Dependency

- Order of construction within a file is known
- Order between files is unspecified!
- Problem when non-local static objects in different files have dependencies.
- A non-local static object is:
  - defined at global or namespace scope
  - declared static in a class
  - defined static at file scope

Static Initialization Solutions

- Just say no -- avoid non-local static dependencies.

- Put static object definitions in a single file in correct order.

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’ in .cpp

Static members

- Static member functions
  - Have no implicit receiver ("this")
    (why?)
  - Can access *only* static member variables
    (or other globals)
  - No ‘static’ in .cpp
  - Can’t be dynamically overridden

To use static members

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

Outline

- Understand the Copy Constructor
- Static in C++
- Strings and Basic I/O

Strings

- In C we used char* to represent a string.
- The C++ standard library provides a common implementation of a string class abstraction named string

Hello World example: From C to C++

```
#include <stdio.h>

void main()
{
        // create string 'str' = "Hello world!"
        char *str = "Hello World!";

        // print sring
        printf("%s\n", str);
}
```

Hello World example: From C to C++

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

int main()
{
        // create string 'str' = "Hello world!"
        string str = "Hello World!";

        cout << str << endl;
        return 0;
}
```

Include header file to use string

string is part of a namespace ("std"), which has to be included (we will be learning more about namespace later)

Different ways to create strings

```
string str = "some text";
```

or

```
string str("some text");
```

Equivalent

or

```
string s1(7,'a');
```

Initialization with size 7 and only a's

or

```
string s2 = s1;
```

Copy constructor

string length

- The length of string is returned by its size() function

```
#include <string>

…

   string str = "something";
   cout << "The size of "
        << str
        << "is " << str.size()
        << "characters." << endl;
```

string length

- In C we had we only had pointers to data
  - Length of string??
- In C++, we have

```
class string {
   …
public:
   …
   unsigned int size();
   …
};
```

String concatenation

- concatenating one string to another is done by the '+' operator
  - Operator-overloading will be seen later

```
string str1 = "Here ";
string str2 = "comes the sun";
string concat_str = str1 + str2;
```

String comparison

- To check if two strings are equal use the '==' operator

```
string str1 = "Here ";
string str2 = "comes the sun";

if ( str1 == str2 )
    /* do something */
else
    /* do something else */
```

String assignment

- To assign one string to another use the “=“ operator.

```
string str1 = “ShanghaiTech”;
string str2 = “SIST”;
str2 = str1;
```

- Now : str2 equals “Sgt. Pappers”

More string functions

- Can check if string is empty

```
bool isEmpty = str1.empty();
```

- Can access single character like C-style string

```
str2[0] = 'a';
```

- Can access single character like C-style string

```
string substring = str1.substr(0,8);
// substring will be "Shanghai"
```

More string functions

- Find a substring inside another string

```
int index = str1.find(substring);
```

  – index will be the starting index of the found substring

- Replace a substring with something else

```
str1.replace(
    index,
    substring.length(),
    newStr );
```

Working with Input/Output in C++

- at top of each file that uses input/output

```
using namespace std;
```

- to use streams to interact with user/console, must have

```
#include <iostream>
```

Input/Output in C++

```
#include <stdio.h>


printf("test: %d\n", x);


scanf("%d", &x);
```

Input/Output in C++

```
#include <stdio.h>
#include <iostream>

printf("test: %d\n", x);


scanf("%d", &x);
```

Input/Output in C++

```
#include <stdio.h>
#include <iostream>
using namespace std;
printf("test: %d\n", x);


scanf("%d", &x);
```

Input/Output in C++

```
#include <stdio.h>
#include <iostream>
using namespace std;
printf("test: %d\n", x);
cout << "test: " << x << endl;

scanf("%d", &x);
```

Input/Output in C++

```
#include <stdio.h>
#include <iostream>
using namespace std;
printf("test: %d\n", x);
cout << "test: " << x << endl;

scanf("%d", &x);
cin >> x;
```

The << Operator

- insertion operator → used along with **`cout`**

- separate each "type" of thing we print out

```
int x = 3;
cout << "X is: " << x
<< "; squared "
<< x*x << endl;
```

The >> Operator

- extraction operator → used with **`cin`**
  - returns a boolean for (un)successful read

- like scanf and fscanf, skips leading whitespace, and stops reading at next whitespace

- don't need to use ampersand on variables

```
cin >> firstName >> lastName >> age;
```