Object – Oriented Programming

Week 7, Spring 2009

Copy Ctor

Weng Kai

http://fm.zju.edu.cn

Wedsneday, April 1, 2009

Quiz

- For the code below

```
void f() {
    Stash students();
    …
}
```

Which statement is RIGHT for the line in function f()?

1. This is a variable definition, while students is an object of Stash, initialized w/ default ctor.
2. This is a function prototype, while students is a function returns an object of Stash.
3. This is a function call.
4. This is illegal in C++.

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 better refer to non-local variables!

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

Example

```
main() {
  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

Const reference parameters

- What if you don’t want the argument changed?
- Use *const* modifier

```
// 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

const in Function returns

- return by const value
  - for user defined types, it means "prevent use as an lvalue"
  - for built-in's it means nothing
- return by const pointer or reference
  - depends on what you want your client to do with the return value

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

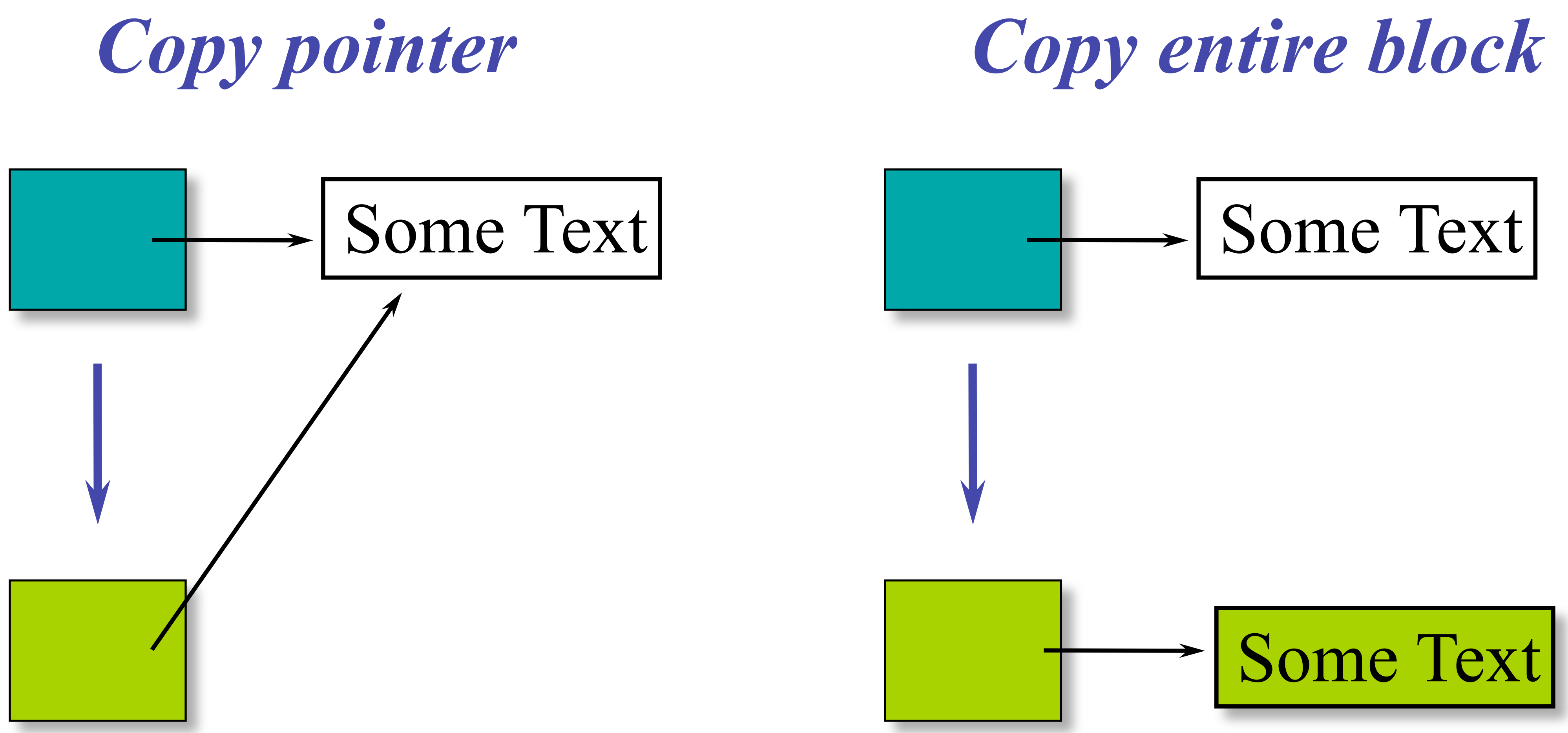
  `T::T(const T&);`
  - Call-by-reference is used for the explicit argument
- C++ builds a copy ctor for you if you don't provide one!
  - Copies each member variable
    - Good for numbers, objects, arrays
  - Copies each pointer
    - Data may become shared!
- Example: HowMany2.cpp

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
};
```

See: Person.h, Person.cpp

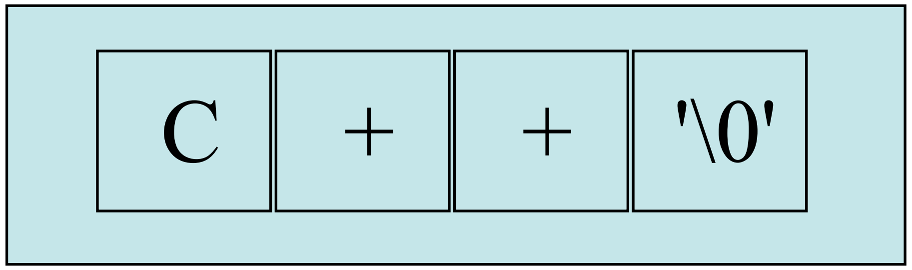
Choices



Copy pointer
Some Text
Copy entire block
Some Text
Some Text

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:

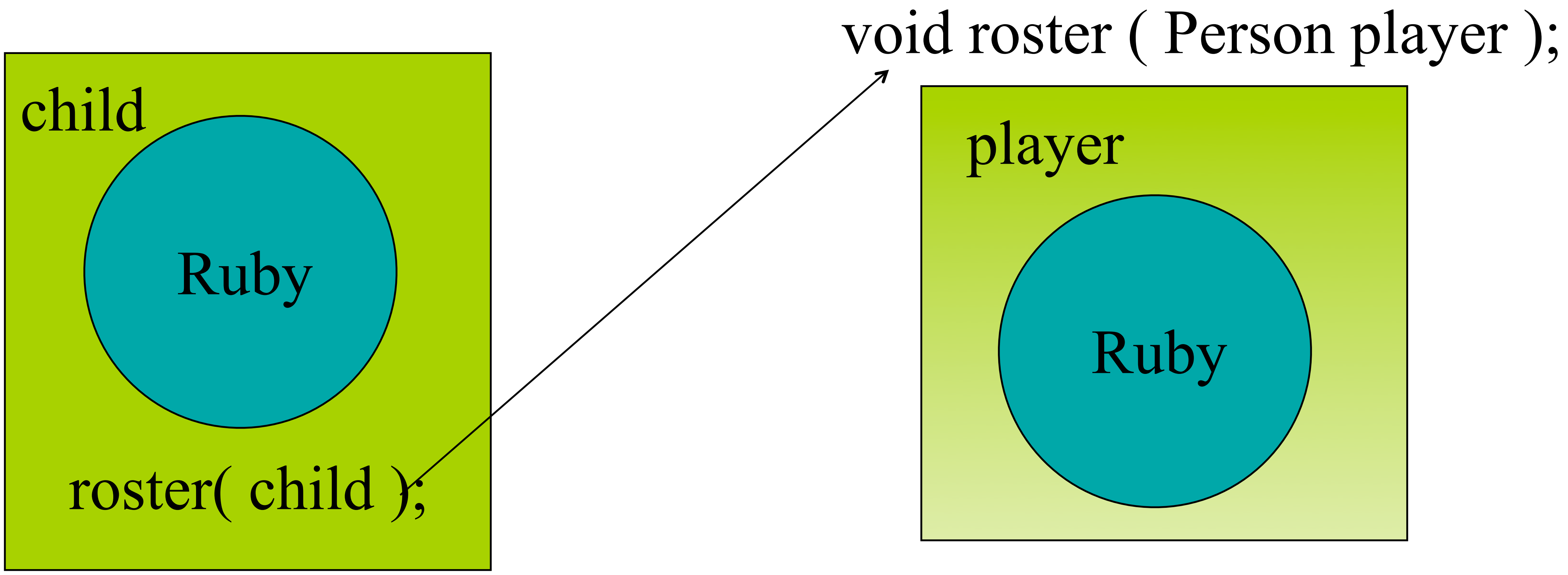
```
Person::Person( const Person& w ) {
    name = new char[::strlen(w.name) + 1];
    ::strcpy(name, w.name);
}
```

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

When are copy ctors called?

- During call by value

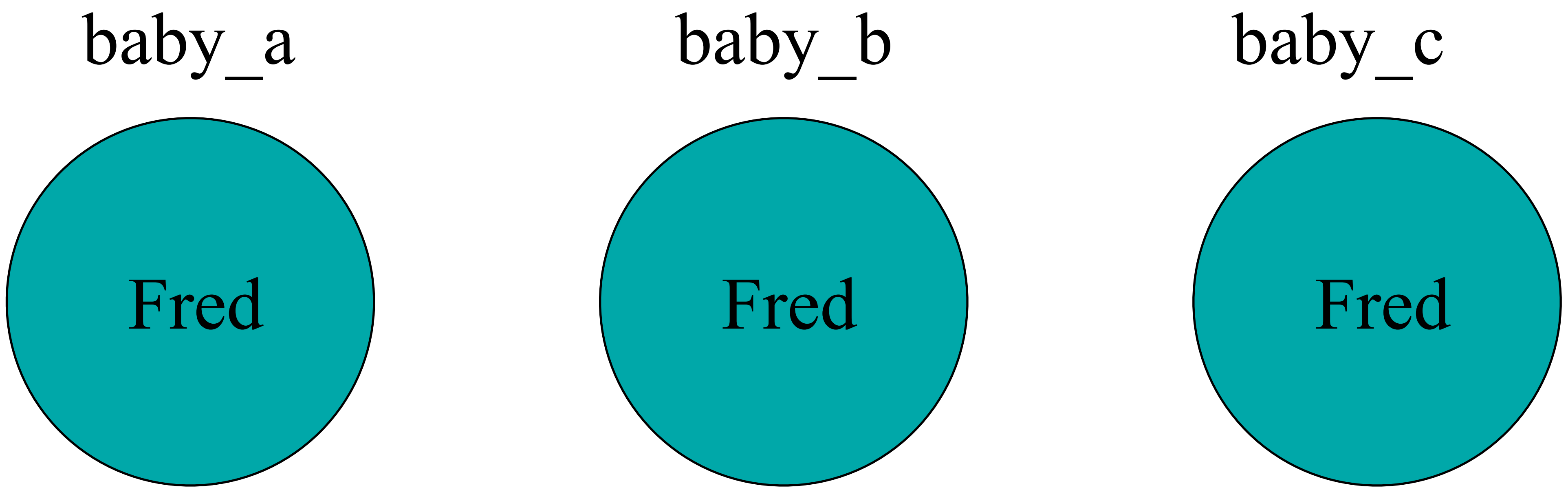
```
void roster( Person );          // declare
  function
Person child( "Ruby" );         // create object
roster( child );            // call function
```



When are copy ctors called?

- During initialization

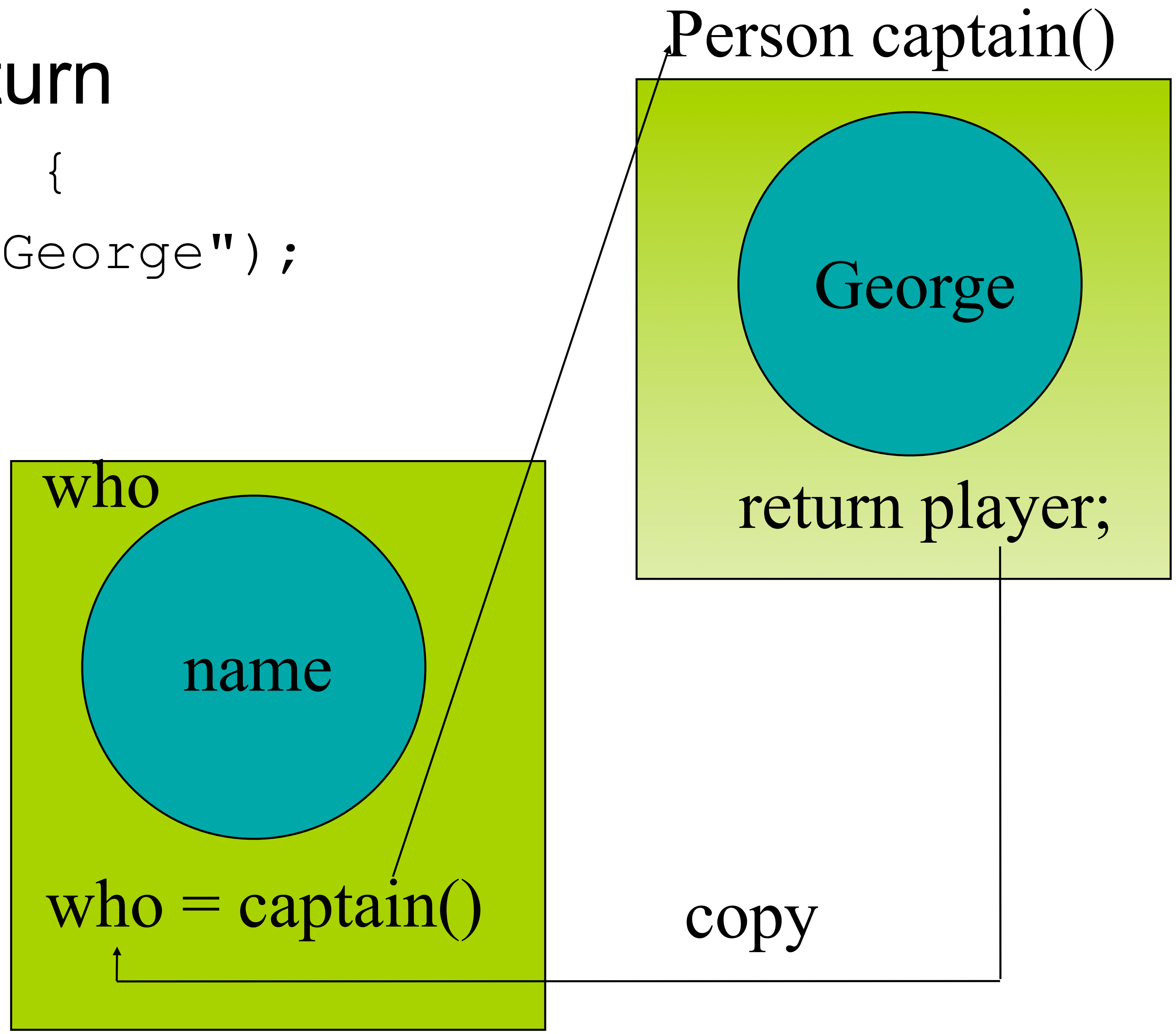
```
Person baby_a("Fred");
// these use the copy ctor
Person baby_b = baby_a;    // not an assignment
Person baby_c( baby_a );   // not an assignment
```



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

- Example: DefaultCopyConstructor.cpp

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 defintion
- Example: NoCopyConstruction.cpp

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

File1

```
int g_global;
static int s_local;

void
func() {
   ...
}

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

?

File2

```
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"
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

- Example: StatMem.h, StatMem.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
- Example: StatFun.h, StatFun.cpp

To use static members

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

Controlling names:

- Controlling names through scoping
- We've done this kind of name control:

```
class Marbles {
   enum Colors { Blue, Red, Green };
   ...
};

class Candy {
   enum Colors { Blue, Red, Green };
   ...
};
```

Avoiding name clashes

- Including duplicate names at global scope is a problem:

```
// old1.h
  void f();
  void g();

// old2.h
  void f();
  void g();
```

Avoiding name clashes (cont)

- Wrap declarations in namespaces.

```
// old1.h
namespace old1 {
  void f();
   void g();
}
// old2.h
namespace old2 {
   void f();
   void g();
}
```

Namespace

- Expresses a logical grouping of classes, functions, variables, etc.
- A namespace is a scope just like a class
- Preferred when only name encapsulation is needed

```
namespace Math {
    double abs(double );
    double sqrt(double );
    int trunc(double);
    ...
  }       // Note: No terminating end colon!
```

Defining namespaces

- Place namespaces in include files:

```
// Mylib.h
namespace MyLib {
  void foo();
  class Cat {
  public:
    void Meow();
  };
}
```

Defining namespace functions

- Use normal scoping to implement functions in namespaces.

```
// MyLib.cpp
#include ”MyLib.h”

void MyLib::foo() { cout << "foo\n"; }
void MyLib::Cat::Meow() { cout << "meow
  \n"; }
```

Using names from a namespace

- Use scope resolution to qualify names from a namespace.
- Can be tedious and distracting.

```
#include ”MyLib.h”
 void main()
 {
     MyLib::foo();
     MyLib::Cat c;
     c.Meow();
 }
```

Using-Declarations

- Introduces a local synonym for name
- States in one place where a name comes from.
- Eliminates redundant scope qualification:

```
void main() {
    using MyLib::foo;
    using MyLib::Cat;
    foo();
    Cat c;
    c.Meow();
}
```

Using-Directives

- Makes *all* names from a namespace available.
- Can be used as a notational convenience.

```
void main() {
    using namespace std;
    using namespace MyLib;
    foo();
    Cat c;
    c.Meow();
    cout << “hello” << endl;
}
```

Ambiguities

- Using-directives may create *potential* ambiguities.
- Consider:

```
// Mylib.h
 namespace XLib {
   void x();
   void y();
 }
 namespace YLib {
   void y();
   void z();
 }
```

Ambiguities (cont)

- Using-directives only make the names available.
- Ambiguities arise only when you make calls.
- Use scope resolution to resolve.

```
void main() {
    using namespace XLib;
    using namespace YLib;
    x(); // OK
    y(); // Error: ambiguous
    XLib::y(); // OK, resolves to XLib
    z(); // OK
}
```

Namespace aliases

- Namespace names that are too short may clash
- names that are too long are hard to work with
- Use aliasing to create workable names
- Aliasing can be used to version libraries.

```
namespace supercalifragilistic {
  void f();
}
namespace short = supercalifragilistic;
short::f();
```

Namespace composition

- Compose new namespaces using names from other ones.
- Using-declarations can resolve potential clashes.
- Explicitly defined functions take precedence.

```
namespace first {
    void x();
    void y();
}
namespace second {
    void y();
    void z();
}
```

Namespace composition (cont)

```
namespace mine {
  using namespace first;
  using namespace second;
  using first::y(); // resolve clashes to first::x()
  void mystuff();
  ...
}
```

Namespace selection

- Compose namespaces by selecting a few features from other namespaces.
- Choose only the names you want rather than all.
- Changes to “orig” declaration become reflected in “mine”.

```
namespace mine {
  using orig::Cat; // use Cat class from orig
  void x();
  void y();
}
```

Namespaces are open

- Multiple namespace declarations add to the same namespace.
  - –Namespace can be distributed across multiple files.

```
//header1.h
namespace X {
  void f();
}
// header2.h
namespace X {
  void g(); // X how has f() and g();
}
```