Copy Ctor

Weng Kai

problem

- 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++.

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&);
  ```

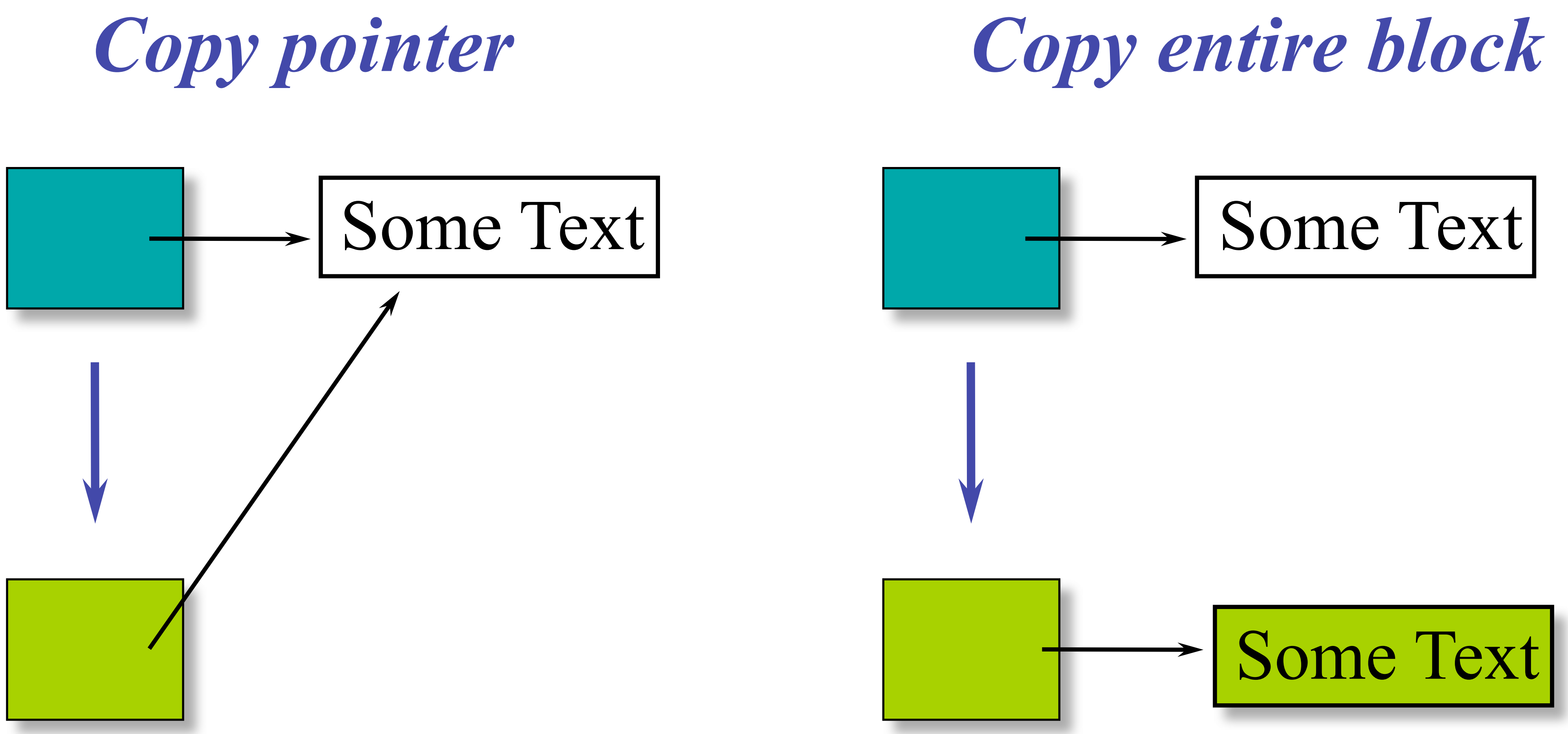
  - Call-by-reference is used for the explicit argument
- C++ builds a copy ctor for you if you don't provide one!
  - Copies each member variable
    - Good for numbers, objects, arrays
  - Copies each pointer
    - Data may become shared!
- Example: HowMany2.cpp

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
};
```

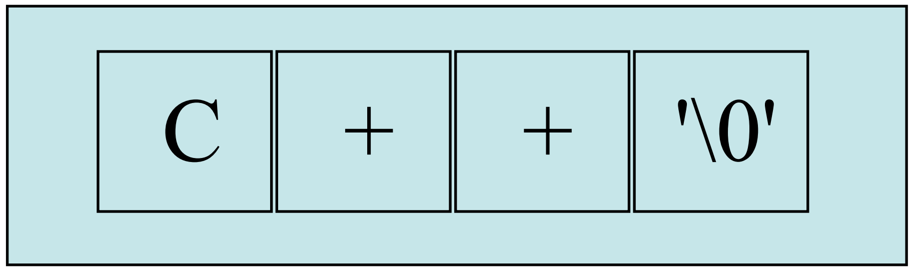
See: Person.h, Person.cpp

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:

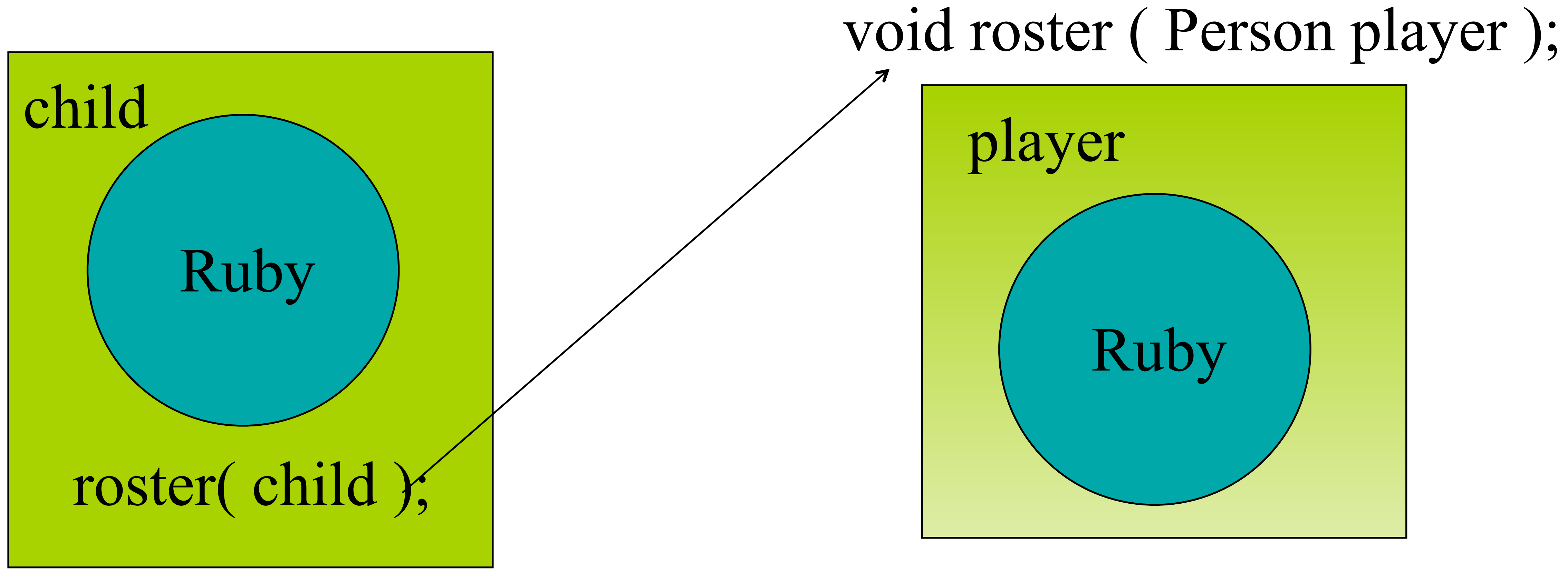
```
Person::Person( const Person& w ) {
    name = new char[::strlen(w.name) + 1];
    ::strcpy(name, w.name);
}
```

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

When are copy ctors called?

- During call by value

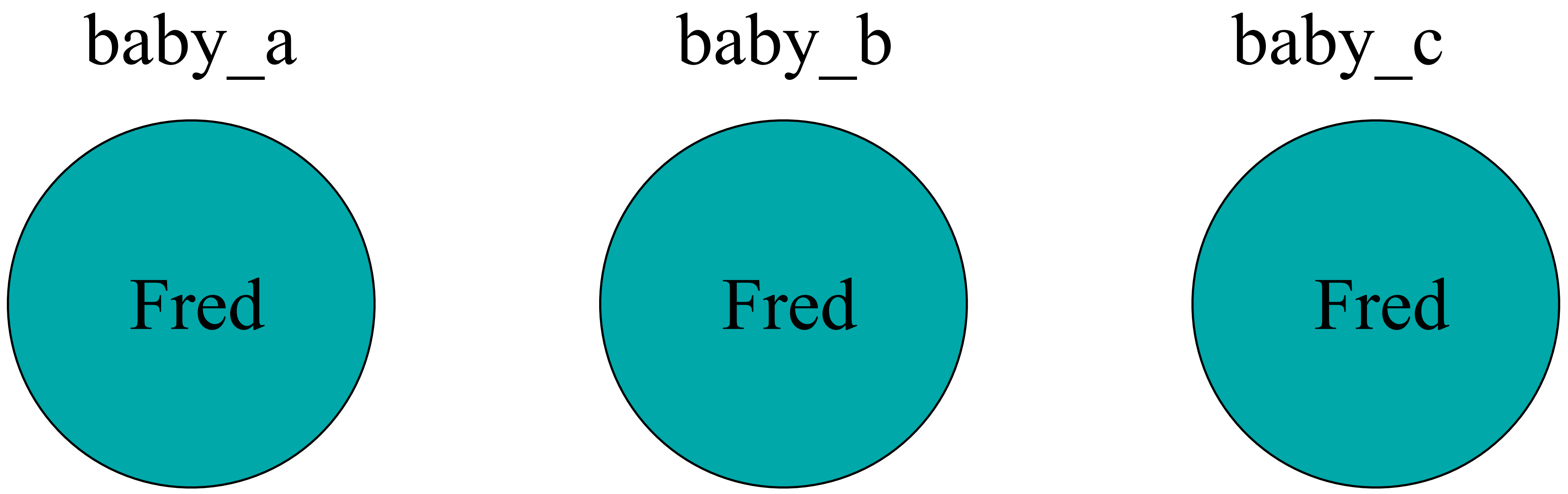
```
void roster( Person );          // declare
  function
Person child( "Ruby" );         // create object
roster( child );             // call function
```



When are copy ctors called?

- During initialization

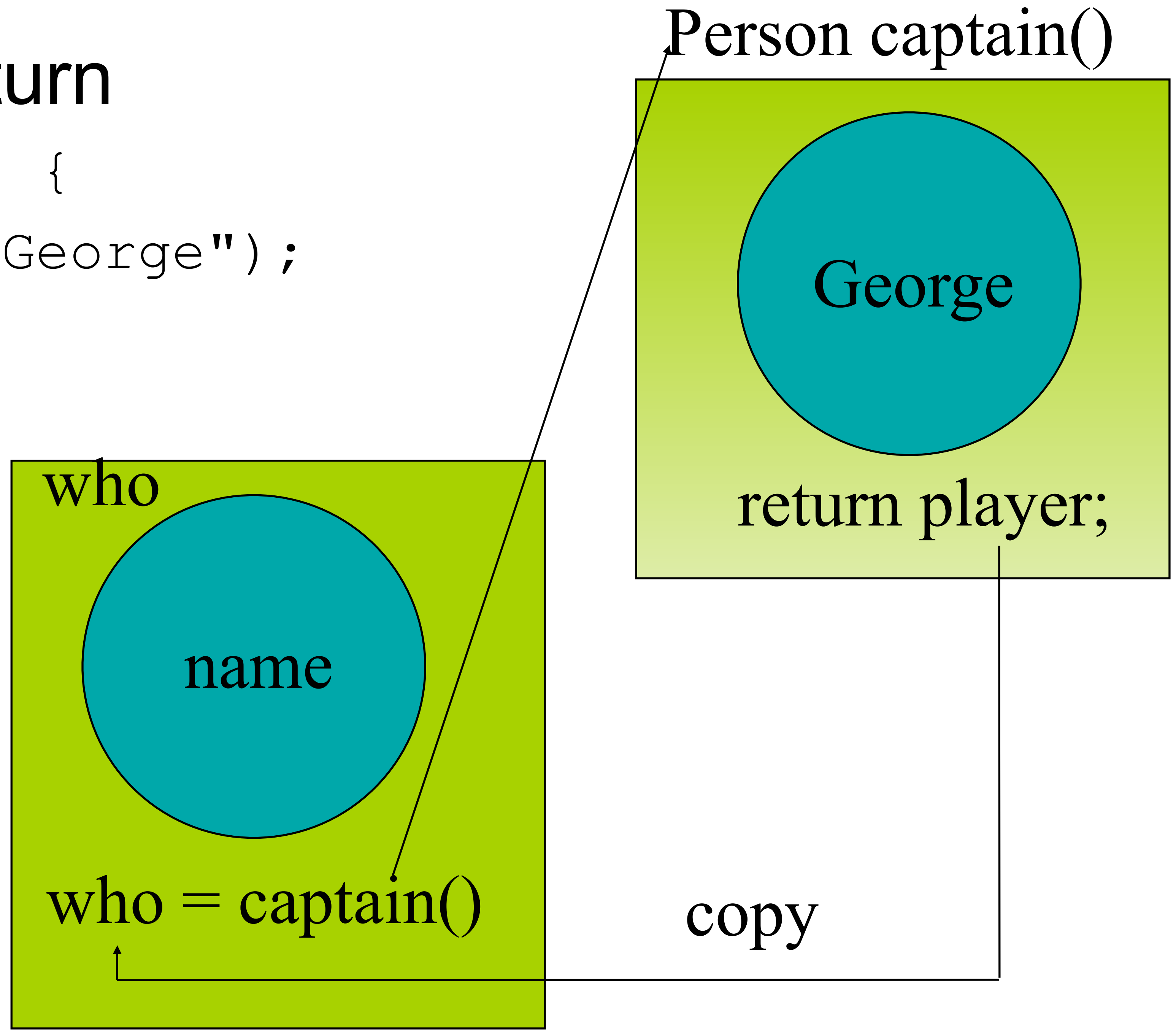
```
Person baby_a("Fred");
// these use the copy ctor
Person baby_b = baby_a;    // not an assignment
Person baby_c( baby_a );   // not an assignment
```



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 copy 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

Overloaded Operators

Overloading Operators

- Allows user-defined types to act like built in types
- Another way to make a function call.

Overloaded operators

Unary and binary operators can be overloaded:

```
+   -   *   /   %   ^    &   |    ~
=   <   >  +=   -=   *=   /=   %=
^=  &=  |=  <<  >>   >>=  <<=  ==
!=  <=  >=  !  &&  ||  ++   --
,   ->*  ->  ()   []
operator new        operator delete
operator new[]    operator delete[]
```

Operators you can't overload

.   .*   ::   ?:

sizeof   typeid

static_cast dynamic_cast const_cast

reinterpret_cast

Restrictions

- Only existing operators can be overloaded (you can't create a ** operator for exponentiation)
- Operators must be overloaded on a class or enumeration type
- Overloaded operators must
  - Preserve number of operands
  - Preserve precedence

C++ overloaded operator

- Just a function with an operator name!
  - Use the `operator` keyword as a prefix to name

    `operator` `*` (…)
- Can be a member function
  - Implicit first argument

    ```
    const String String::operator +(const String& that);
    ```
- Can be a global (free) function
  - Both arguments explicit

    ```
    const String operator+(const String& r, const String& l);
    ```

How to overload

- As member function
  - Implicit first argument
  - No type conversion performed on receiver
  - Must have access to class definition

Operators as member functions

```
class Integer {
public:
  Integer( int n = 0 ) : i(n) {}
  const Integer operator+(const Integer& n) const{
      return Integer(i + n.i);
  }
  ...
private:
   int i;
};
```

See: OperatorOverloadingSyntax.cpp

Member Functions

```
Integer x(1), y(5), z;
x + y;    ====> x.operator+(y);
```

- Implicit first argument
- Developer must have access to class definition
- Members have full access to all data in class
- No type conversion performed on receiver

```
    z = x + y;   √
    z = x + 3;   √
```
~~`z = 3 + y;`~~

Member Functions...

- For binary operators (+, -, *, etc) member functions require one argument.
- For unary operators (unary -, !, etc) member functions require no arguments:

```
 const Integer operator-() const {
    return Integer(-i);
}
...
z = -x;  // z.operator=(x.operator-());
```

How to overload

- As a global function
  - Explicit first argument
  - Type conversions performed on both arguments
  - Can be made a friend

Operator as a global function

```
const Integer operator+(
            const Integer& rhs,
            const Integer& lhs);
Integer x, y;
x + y     ====> operator+(x, y);
```

- Explicit first argument
- Developer does not need special access to classes
- May need to be a friend
- Type conversions performed on both arguments

Global operators (friend)

```
class Integer {
  friend const Integer operator+ (
             const Integer& lhs,
             const Integer& rhs);
   ...
}
const Integer operator+(
            const Integer& lhs,
            const Integer& rhs){
   return Integer( lhs.i + rhs.i );
}
```

Global Operators

- binary operators require two arguments
- unary operators require one argument
- conversion:

```
z = x + y;
z = x + 3;
z = 3 + y;
z = 3 + 7;
```

- If you don't have access to private data members, then the global function must use the public interface (e.g. accessors)

Tips:Members vs. Free Functions

- Unary operators should be members
- = () [] -> ->* must be members
- assignment operators should be members
- All other binary operators as non-members

Argument Passing

- if it is read-only pass it in as a const reference (except built-ins)
- make member functions const that don't change the class (boolean operators, +, -, etc)
- for global functions, if the left-hand side changes pass as a reference (assignment operators)

Return Values

- Select the return type depending on the expected meaning of the operator. For example,
  - For operator+ you need to generate a new object. Return as a const object so the result cannot be modified as an lvalue.
  - Logical operators should return bool (or int for older compilers).

The prototypes of operators

- +-*/%^&|~
  - const T operatorX(const T& l, const T& r);
- ! && || < <= == >= >
  - bool operatorX(const T& l, const T& r);
- []
  - E& T::operator[](int index);

operators ++ and --

- How to distinguish postfix from prefix?
- postfix forms take an int argument -- compiler will pass in 0 as that int

```
 class Integer {
public:
   ...
   const Integer& operator++();   //prefix++
   const Integer operator++(int); //postfix++
   const Integer& operator--();   //prefix--
   const Integer operator--(int); //postfix--
   ...
};
```

Operators ++ and --

```
const Integer& Integer::operator++() {
   *this += 1;      // increment
   return *this;    // fetch
}

// int argument not used so leave unnamed so
// won't get compiler warnings
const Integer Integer::operator++( int ){
   Integer old( *this );   // fetch
   ++(*this);              // increment
   return old;             // return
}
```

Using the overloaded ++ and --

```
// decrement operators similar to increment
    Integer x(5);
    ++x;
            // calls x.operator++();
    x++;
            // calls x.operator++(0);
    --x;
            // calls x.operator--();
    x--;
            // calls x.operator--(0);
```

- User-defined prefix is more efficient than postfix.

Relational operators

- implement != in terms of ==
- implement >, >=, <= in terms of <

```
class Integer {
     public:
        ...
     bool operator==( const Integer& rhs ) const;
     bool operator!=( const Integer& rhs ) const;
     bool operator<( const Integer& rhs ) const;
     bool operator>( const Integer& rhs ) const;
     bool operator<=( const Integer& rhs ) const;
     bool operator>=( const Integer& rhs ) const;
  }
```

Relational operators

```
bool Integer::operator==( const Integer& rhs ) const {
   return i == rhs.i;
}
// implement lhs != rhs in terms of !(lhs == rhs)
bool Integer::operator!=( const Integer& rhs ) const {
   return !(*this == rhs);
}

bool Integer::operator<( const Integer& rhs ) const {
   return i < rhs.i;
}
```

Relational Operators...

```
// implement lhs > rhs in terms of lhs < rhs
bool Integer::operator>( const Integer& rhs ) const {
   return rhs < *this;
}
// implement lhs <= rhs in terms of !(rhs < lhs)
bool Integer::operator<=( const Integer& rhs ) const {
   return !(rhs < *this);
}
// implement lhs >= rhs in terms of !(lhs < rhs)
bool Integer::operator>=( const Integer& rhs ) const {
   return !(*this < rhs);
}
```

Operator []

- Must be a member function
- Single argument
- Implies that the object it is being called for acts like an array, so it should return a reference

```
Vector v(100);   // create a vector of size 100

v[10] = 45;
```

(Note: if returned a pointer you would need to do:

```
*v[10] = 45;
```

See: vector.h, vector.cpp