Controlling copies of objects

Copying an object is not always the same as copying its contents

The issue

- A C++ class defines the meaning of every operation performed on objects of that class.
- If you don't define certain operations, the compiler does it for you:
  - Ordinarily, copying a class object means copying its elements
  - That behavior is often inappropriate for classes that represent abstract data types

Objects versus values

- What is the difference?
- Why does it matter?
- How do we model the difference?
- *Warning: These concepts are somewhat vague and approximate, and not everyone agrees on what they mean*

Values

- Once created (computed), they are never modified
- No way to tell the difference between a value and a copy of it
- Typically called *rvalues* in C and C++

Objects

- Referred to by *lvalues* in C and C++
- Once created, they can be modified
- A copy of an object is distinct from the original
  - Modifying one does not modify the other
  - Their addresses are different (whatever that means in a given language)
- Objects usually *contain* values

Examples

- 3 is a value
- If we define a variable, as in

```
int x = 3;
```

that variable is an object, which contains the value 3

Variables are objects

- We can demonstrate that distinct variables are distinct objects by changing one of them and observing that the other does not change
- Even if the variables are `const`, we can still observe that they have different addresses

Pointers are values

- A copy of a pointer is indistinguishable from the original, even though pointers identify (*i.e.* contain the addresses of) objects
- Variables that contain pointers are objects, as usual

Arrays are (essentially) objects

- In C and C++, the name of an array is usually converted to the address of its initial element, which is distinct for distinct arrays
- By implication, string literals (such as “`abc`”) are objects, not values, because they are arrays

A concrete example

- We will define a class `String` whose objects represent variable-length strings of characters
- We would like `String` objects to behave much like values
- In particular, we would like to be able to pass `String`s as arguments, return them as results, etc.

One implementation snag

- A string literal, as built into the C and C++ languages, is an object, but it evaluates to a pointer, which is a value
- That means that “copying” a string literal copies the pointer, which results in two pointers that identify the same object

Literals and aliasing

- Suppose we say

```
char* x = new char[4];
strcpy(x, “cat”);
char* y = x;
x[2] = ‘r’;
```

Then x and y refer to the same object, so changing x[2] changes y[2] also

- This behavior makes it hard to treat strings as values

Strings as values

- What we would like is an abstraction that lets us use strings as if they were values:
  - Copying a string should copy the characters that constitute it
  - Freeing a string should free its characters
- To define such an abstraction, we need to be able to define copying

What is copying?

- Copying an object creates a copy of it
- Therefore, copying is a way of constructing a new object
- Accordingly, we say how to copy objects of a particular class by writing a *copy constructor* for that class

What is a copy constructor?

- Suppose we have an object of class X and we want to construct another object of class X from it
- Then we need a constructor that takes an object of class X as argument

Overloading constructors

- The copy constructor had better not be the only way to construct an object, because if it were, there would be no way to create the first object
- Therefore, classes that have a copy constructor will invariably have more than one constructor

First try

- It might seem that we could define a copy constructor this way:

```
class X {
public:
    X(X);       // copy constructor?
    // …
};
```

- However, this strategy fails hideously

Why X(X) doesn't work

- Recall that passing an argument to a function copies the argument
- Therefore, calling X(X) must copy the object being copied before it can copy it
  - To do that, it would have to use the copy constructor, but calling the copy constructor must first copy the argument
    - To do that, it would have to use the copy constructor, but ...

What do we really want

- To copy an object, we want to run a copy constructor whose parameter is bound to that object *without copying it*
- Moreover, we do not want to modify the original object in order to copy it
- Therefore, we want the copy constructor to take a reference to `const` as its parameter

Writing a copy constructor

```
class String {
public:
    // …
    String(const String&);
    // …
};

String::String(const String&)
{ /* … */ }
```

What operations should a `String` support?

- Create a `String` from a null-terminated character array
- Destroy a `String`
- Copy a `String`
- Print a `String`

We can start coding

```
Class String {
    friend ostream& operator<<
        (ostream&, const String&);
public:
    String();       // empty string
    String(const char*);
    String(const String&);

private:
    char* data;
};
```

Default constructor

- Necessary in order to allow

```
String s;
```

or

```
String s[10];
```

- We will allocate a null string:

```
String::String(): data(new char[1])
{
    data[0] = '\0';
}
```

Construct a `String` from a character array

```
String::String(const char* s):
    data(new char[strlen(s) + 1])
{
    strcpy(data, s);
}
```

The copy constructor

```
String::String(const String& s):
    data(new char[strlen(s.data)+1])
{
    strcpy(data, s.data);
}
```

The rest of it

```
String::~String()
{
    delete[] data;
}

ostream& operator<<
    (ostream& o, const String& s)
{
    o << s.data;
    return o;
}
```

Example

```
int main() {
    String hello(“Hello ”);
    String world(“world”);
    cout << hello;
    cout << world << endl;
}
```

Two problems

- Sometimes we will copy strings when we'd rather not; this problem affects performance but not correctness
- We still haven't defined the meaning of

```
String s1, s2;
s1 = s2;        // What does this do?
```

Assignment is not copying

- It might appear that

```
s1 = s2;
```

  makes s1 into a copy of s2, but that reasoning is deceptive
- The reason is that s1 already had a value, and we must first dispose of it somehow
- Also, how do we specify assignment?

Defining assignment

- C++ treats assignment as a separate operation from copying
- Assignment is a member function with the strange name of `operator=`
- It should return a reference to the left-hand side, for consistency with built-in assignment

Example of assignment

```
class String {
public:
    // …
    String& operator=(const String&);
    // …
};
```

Assignment usually has three parts

- Check whether the left-hand and right-hand sides are the same object
  - This is not just for efficiency; we must avoid deleting the object's contents and then trying to assign them!
- Do the assignment (often like executing the destructor and copy constructor)
- Return the left-hand side

Referring to the present object

- Within the body of a member function, the keyword `this` is a pointer to the object that is currently in use
- Therefore, the expression `*this` is a reference to the present object
- Assignment operators will therefore usually say

```
return *this;
```

Putting it all together

```
String&
String::operator=(const String& s)
{
  if (this != &s) {
    delete[] data;
    data = new char[strlen(s.data)+1];
    strcpy(data, s.data);
  }
  return *this;
}
```

Regrouping modules

- There are four interface operations
  - Construct from a character array
  - Construct from a(nother) String
  - Assign
  - Destroy
- …but only two in implementation
  - Copy in a character array
  - Destroy

Implementation subroutines

- We can't call constructors explicitly, and shouldn't call destructors, but we can regroup their work into auxiliary functions
  - Copy in a string with `init`
  - Delete our data with `destroy`
- The other operations will call these

Revise the class

```
class String {
    friend ostream& operator<<
        (ostream&, const String&);
public:
    String();
    String(const char*);
    String(const String&);
    String& operator=(const String&);
    ~String();
private:
    char* data;
    void init(const char*);
    void destroy();
};
```

Now we can initialize and destroy once

```
void String::init(const char* s)
{
    data = new char[strlen(s) + 1];
    strcpy(data, s);
}

void String::destroy()
{
    delete[] data;
}
```

The other operations become easier

```
String::String()
{
    init(“”);
}

String::String(const char* s)
{
    init(s);
}
```

More operations

```
String::String(const String& s)
{
    init(s.data);
}

String::~String()
{
    destroy();
}
```

Assignment

```
String&
String::operator=(const String& s)
{
    if (this != &s) {
        destroy();
        init(s.data);
    }
    return *this;
}
```

Where are we now?

- We know how to define the meaning of copying and assignment for classes
- We used that tool to define a class that behaves like a variable-length string

The next couple of weeks

- Proposals due ***this Friday***
  - see notes from lecture 2 for details
  - no homework this week so you can focus on the presentations
- Presentations in class next week
- Midterm Wednesday, March 10