Handles and use counts

Making objects act like values

Overview

- Suppose we have a class whose objects we do not wish to copy
  - strings
  - arrays and other containers
- How can we avoid copying such objects except when truly necessary?
- One way: Copy something else instead

What is a copy?

- A copy of an object is a distinct object with the same properties as the original
- If the object is part of a large data structure, we might distinguish between
  - copying just that object (shallow copy)
  - copying the whole structure (deep copy)

Why do we want to copy objects?

- How do you tell whether two names refer to the same object?
  - Modify one object
  - Observe the change in the other
- Making a copy of an object
  - usually precedes a change to one of the objects
  - is unnecessary otherwise

Recall our `String` class

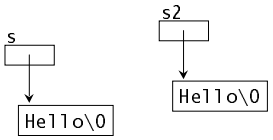
```
class String {
    friend ostream& operator<<
        (ostream&, const String&);
public:
    String();
    String(const char*);
    String(const String&);
    String& operator=(const String&);
    ~String();
private:
    char* data;
    void init(const char*);
    void destroy();
};
```

`Strings` are treated as values

- Copying a `String` copies the characters that constitute it
- Freeing the `String` frees its characters

An example of how `Strings` work

```
String s = "Hello";
String s2 = s;
```



Copy and assignment…

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

…result in new data being allocated

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

Often these allocations are unnecessary

```
String f(String s)
{
   String x;

   // …

   return x;
}
```

makes an extra copy of `s` and perhaps of `x` as well.

Handles

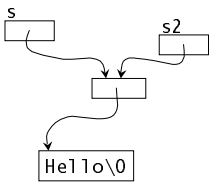
- How can we avoid copying such objects except when truly necessary?
- One way: copy something else instead
- Each `String` will be a *handle class*
- class `String` and its implementation class will cooperate to manage the data and avoid unnecessary copies

The implementation class

- Will hold the data pointer and an `int` that will count how many `Strings` point to that object
- When `Strings` are copied, we'll increment the use count
- When they are destroyed will decrement the use count
- When the last `String` is destroyed we'll free the characters

The revised data structure

```
String s = "Hello";
String s2 = s;
```



String_rep class

```
class String_rep {
    char* data;
    int use;
    friend class String;
};
```

Revised String class

```
class String {
    friend ostream& operator<<
        (ostream&, const String&);
    String_rep* r;
public:
    String();
    String(const char*);
    String(const String&);
    String& operator=(const String&);
    ~String();
};
```

The String_rep operations

```
class String_rep {
    friend class String;
    friend ostream& operator<<
        (ostream&, const String&);

    String_rep(unsigned n):
        data(new char[n]), use(1) { }
    ~String_rep()
        { delete [] data; }

    char* data;
    int use;
};
```

The String constructors

- Allocate a new String_rep
- Copy the characters

```
String::String(): r(new String_rep(1))
{
    r->data[0] = '\0';
}
String::String(char *p):
    r(new String_rep(strlen(p) + 1)) {
        strcpy(r->data, p);
}
```

Copy Constructor

- Just copies the String_rep pointer and fiddles the use count

```
String::String(const String&s): r(s.r)
{
    ++r->use;
}
```

The destructor

- Checks whether it is the last `String`, and if so, frees the `String_rep`

```
String::~String()
{
    if (--r->use == 0)
        delete r;
}
```

Assignment

- As usual, assignment must guard against self-assignment
- Assignment itself, involves copying the `String_rep` pointer and fiddling use counts
- The data array itself is not copied

Assignment operator

```
String&
String::operator=(const String& s)
{
    ++s.r->use;
    if (--r->use == 0)
        delete r;
    r = s.r;
    return *this;
}
```

Output

- We have to change the output operator to account for the indirection through `r`

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

Operations on `String`

- So far, we can only create and assign `Strings`
- Some operations will involve copying the underlying data, others won't
- For example concatenation

Compound concatenation

- Like the assignment operator, it changes the left-hand-side
- Thus, we'll need to allocate a new `String_rep`

Operator+=

```
String&
String::operator+=(const String& s)
{
  if (s.r->data[0] != '\0') {
    String_rep* newr =
      new String_rep(strlen(r->data) +
                     strlen(s.r->data) + 1);
    strcpy(newr->data, r->data);
    strcat(newr->data, s.r->data);
    if (--r->use == 0)
      delete r;
    r = newr;
  }
  return *this;
}
```

Binary concatenation

- Should be a non-member function
  - += modifies its LHS
  - + does not
- We'd like to allow conversions

```
String world = "world";
String hello = "hello";
String out;

out = "hello " + world;
out = hello + " world";
```

Concatenation

```
String
operator+(const String & lhs, const String& rhs)
{
   String ret = lhs;
   ret += rhs;
   return ret;
}
```

Conclusions

- We can use constructors and destructors to define classes whose objects behave much like values
- We can use use-counted memory allocation to avoid having to copy data needlessly
- These techniques are fundamental to C++ programming

Smalltalk does it differently

- In Smalltalk, copying is always explicit
  - all types are objects
  - all variables are references
  - after x←y, x and y always refer to the same object
- Therefore, the language deliberately prohibits changing the value of objects of types such as `int` and `string`

Java does it differently, too

- Simple types, such as `int`, are values, not objects
  - Each variable of one of these types has its own copy
  - Therefore, operations such as ++ present no problem
- Strings are objects, so changing part of a string presents a problem, so Java prohibits it

ML has a systematic approach

- All values are just that—values—and therefore cannot be changed once created
- For every value type T, there is a corresponding object type `T ref`, values of which type behave similarly to variables in Smalltalk or Java

The root of the problem

- Most programmers think of some types as being values and others as objects
- Languages that come close to treating everything as an object have trouble dealing with values
- C++, which treats almost everything as a value, requires extra awareness to deal with objects

Dealing with objects in C++

- A pointer is a value that is bound to a particular object
- A reference isn't even a value: It's just a name that is bound to an object
- Virtual functions work only through such bindings
- We can define classes that also act like bindings and are useful in other ways

Properties of binding objects

- Attach one to another object
- Access (modify?) the object to which it is attached
- Copy the binding object (which might copy the other object or not)
- Destroy the binding object (which might destroy the other object or not)

Binding objects have many forms and names

- Pointers
- References
- Smart pointers
- Handles
- Surrogates
- Iterators

Reminder: Proposals are due Friday

- The names of the team members
- A description of the project
- What will it "cost" (schedule)
- How will you build it (organization)
- What will it build on? (*i.e.* libraries and other tools)
- Why did you choose this project?

Project Description

- What it will do
- What else it will do if you have time
- Why it is interesting
- What is challenging about it

Presentations

- A sales pitch for the project
- 10 minutes per presentation (so that every team gets a chance)
- It's OK if one team member speaks for the team
- Written handouts and overhead slides (no more than 5) are encouraged