CS107L Handout 02
Autumn 2007 October 5, 2007

Copy Constructor and operator=

Much of the surrounding prose written by Andy Maag, CS193D instructor from years ago.

The compiler will supply a default (zero-argument) constructor if the programmer doesn’t. This default constructor will call each data member’s default constructor in order to fully initialize the object. If the programmer wishes to displace the default constructor, he simply provides one. The same thing goes for the destructor.

There is another type of compiler-synthesized constructor—the **copy constructor**, and it's called whenever an object needs to be constructed from an existing one.

Suppose we had a class to encapsulate strings, and we call this class **`string`**. The details of the dynamic memory management are annoying enough that we might want to abstract those details away.

```
// string.h, take one
class string {

   public:
      string(const char *str = "");
      ~string();

   private:
      initializeFrom(const char *front, const char *end)
      char *front;
      char *end;
};

// string.cc take one
string::string(const char *str)
{
   initializeFrom(str, str + strlen(str));
}

string::~string()
{
   delete[] front;
}

void string::initializeFrom(const char *front, const char *end)
{
   int length = end - front;
   this->front = new char[length + 1];
   this->end = this->front + length;
   strncpy(this->front, front, length + 1);  // include the '\0'
}
```

For this example, we'll assume that the **`string`** constructor allocates space for the characters, and the destructor frees up that same space. Internally, the are two iterator-like points that point to the leading character and the null character at the end. The

The copy constructor may be called when doing simple initializations of a **`string`** object:

```
string professor("Professor Plum");
string clone = professor; // copy constructor gets called.
```

More importantly, the copy constructor is called when passing an object by value, or returning an object by value. For instance, we might have a function that translates a sentence to Pig Latin:

```
static string translate(string sentence)
{
   // tokenizes, rotates, and rebuilds translation
}
```

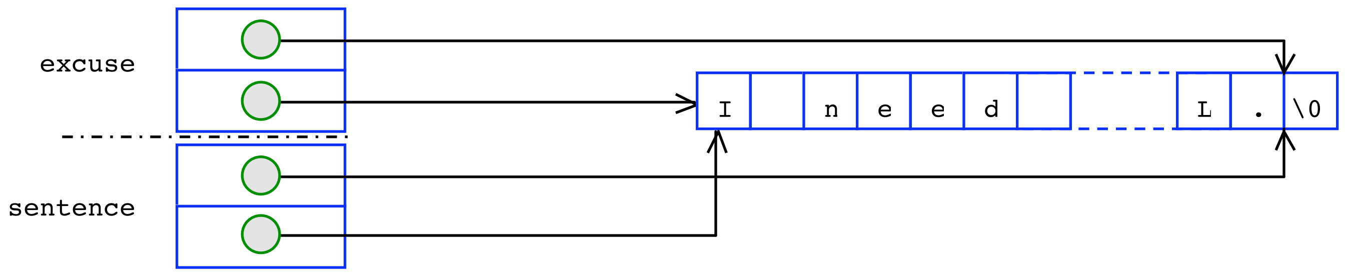
We might declare a **`string`** and call the **`translate`** function like this:

```
string excuse("I need an extension because I forgot I was in CS107L.");
cout << "Translation: \"" << translate(excuse) << "\"" << endl;
```

When passing the sentence object, the copy constructor is called to copy the **`string`** object from the calling function to the local parameter in the **`translate`** function. Because we did not specify a copy constructor, the default copy constructor is called. In this case, it's just not what we want.

Default Copy Constructor

The default copy constructor does a member-wise copy of all the primitive and embedded object fields. For our **`string`** class, the default copy constructor simply copies the two pointers that are embedded inside, as it would for any set of pure primitive types. However, the characters themselves are **not** copied. This is called a **shallow copy**, because it only copies the data down to one level. In memory, what we'd have would look like this:

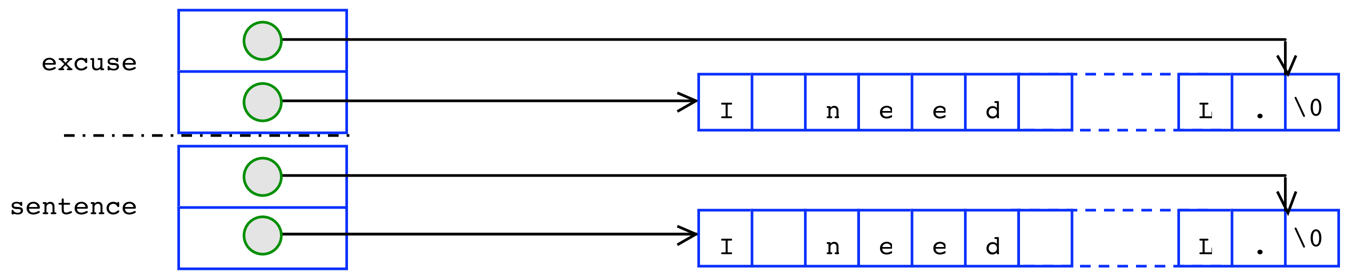


This causes a problem because now the two objects share the same characters data. If we changed any of the characters within the **sentence** object, it would change the characters in the **excuse** as well. What's worse is that after the **translate** function exits, the **string** destructor is called on the **sentence** object, which frees the character array more officially owned by **excuse**. Bad times.

What We Want: The Deep Copy

In order to avoid potential disasters like this, what we want is a **deep copy** of the **string** object. That means we want the copy constructor to synthesize of new character array that's a logical clone of the original, but memory independent of it.

What we want is a picture in memory which looks like this:



Now we can manipulate the characters within the **sentence** object and not affect the original **excuse** object. When the character array within the copy is freed on exit the **translate** function, the original **string** object is still going strong. The copy constructor (be it the bad compiler-provided one or the correct user-defined one) is also invoked when returning an object by value. We didn't even begin to hit on those problems, though they are effectively the same.

Declaring a Copy Constructor

In order to provide deep-copy semantics for our **string** object, we need to declare our own copy constructor. A copy constructor takes a constant reference to an object of the class' type as a parameter and has no return type. We would declare the copy constructor for the **string** within the class declaration like this:

```
// string.h take two
class string {

   public:
      string(const char *str = "");
      string(const string& original);
      ~string();

   private:
      void initializeFrom(const char *front, const char *end);
      char *front;
      char *end;
};
```

We'd implement the copy constructor like this:

```
string::string(const string& original)
{
   initializeFrom(original.front, original.end);
}
```

You should provide a copy constructor for any class for which you want deep copy semantics and the compiler will do the wrong thing. If the class only has data members that are integral types (that is, it contains no pointers or open files) and/or direct objects (which have properly implemented copy constructors), you can omit the copy constructor and let the default copy constructor handle it.

Limitations

The copy constructor is not called when doing an object-to-object assignment. For instance, if we had the following code, we would still only get a shallow copy:

```
string betty("Betty Rubble");    // Initializes string to "Betty Rubble"
string wilma;                    // Initializes string to empty string
wilma = betty;
```

This is because the **assignment operator** is being called instead of the copy constructor. By default, the assignment operator does a member-wise copy of the object, which in this case gives a shallow copy. However, C++ gives us the ability to override the default assignment operator, and we'll learn that today too.

Incidentally, there may be cases when you want to prevent anyone from copying a class object via the copy constructor. By declaring a copy constructor as a **`private`** constructor within the class definition and **not** implementing it, the compiler will prevent the passing of objects of that class by value.

Assignment

It is possible to redefine the **assignment** operator for a class. The assignment operator must be defined as a member function in order to guarantee that the left-hand operand is an object. By default, the assignment operator is defined for every class, and it does a member-wise copy from one object to another. This is a shallow copy, so the assignment operator should generally be redefined for any class that contains pointers. You can declare the assignment operator as a **`private`** operator and not provide an implementation in order to prevent object-to-object assignment.

The key difference between assignment and copy construction is that assignment possibly involves the destruction of embedded resources. We need to overload the assignment operator whenever we want to guarantee that memory (or perhaps other resources) are properly disposed of before they're replaced. The syntax for the assignment method is a trifle quirky, but it's just that: syntax, and you just treat the prototype as boilerplate notation and otherwise implement the method as you would any other.

```
// string.h take three
class string {

   public:
      string(const char *str = "");
      string(const string& original);
      const string& operator=(const string& rhs);
      ~string();

   private:
      void initializeFrom(const char *front, const char *end);
      char *front;
      char *end;
};

// addition to string.cc

const string& string::operator=(const string& rhs)
{
   if (this != &rhs) {
      delete[] front;
      initializeFrom(rhs.front, rhs.end);
   }

   return *this;
}
```

A good way to look at the assignment operator: Think of it as destruction followed by immediate reconstruction. The **`delete[]`** would appear in the destructor as well, whereas the next three lines are very constructor-ish.

Caveats 1: The **`this != &rhs`** checks to see whether we're dealing with a self-assignment. An expression of the form **`me = me`** is totally legal, but there's no benefit in actually destroying the object only to reconstruct it only to take the same form. In fact, the **`delete[]`** line would actually release the very memory that we need on the very next line.

Caveat 2: The return type is **`const string&`**, and the `return` statement is **`return *this`**, because cascaded assignments involving **`string`**s should behave and propagate right to left just as they would with **`int`**s or **`double`**s.

```
string heroine("ginger");
string cluck("babs");
string quack("rhodes");
string meanie("mrs. tweety");

meanie = quack = cluck = heroine;
cout << heroine << cluck << quack << meanie << endl;
// should print: gingergingergingerginger
```

Problem

Consider the following C++ **`struct`** definitions:

```
struct scooby {

  scooby(const string& jones) { fred = jones; daphne = new string(fred); }
  scooby(const scooby& blake) : daphne(NULL), fred(blake.fred) {}
  ~scooby() { delete daphne; }

  string fred;
  string *daphne;
};

struct dooby {

  dooby(const string& shaggy,
        const string velma) : doo(velma), scrappy(shaggy) {}

  const string scrappy;
  scooby doo;
};
```

Assume that the **`whereareyou`** function has just been called. Specify the ordering for all the calls to the **`string`** memory management routines: the zero-argument constructor, the (**`char*`**) constructor, the copy constructor, the **`operator=`** assignment operator, and the destructor.

```
static void whereareyou(const string& astro, const string& dino)
{
   dooby lassie(astro, dino);
   dooby tiger(lassie);
}
```

Answer:

- copy constructor initializing **`lassie.velma`** out of **`dino`**
- copy constructor initializing **`lassie.scrappy`** out of **`shaggy`**
- zero arg constructor initializing **`lassie.doo.fred`**
- **`operator=`** reassigning **`lassie.doo.fred`** to be a clone of **`jones`**
- copy constructor initializing a dynamically allocated **`string`** (address assigned to **`lassie.doo.daphne`**)
- destructor to dispose of local paramter **`velma`**
- copy constructor initializing **`tiger.scrappy`** to be a clone of **`lassie.scrappy`**
- copy constructor for **`tiger.doo.fred`**
- destructor disposing of **`tiger.doo.fred`**
- destructor disposing of **`tiger.scrappy`**
- destructor disposing of heap-based **`string`** pointed to by **`lassie.doo.daphne`**
- destructor disposing of **`lassie.doo.fred`**
- destructor disposing of **`lassie.scrappy`**