CS 213 -- Lecture #10

"Late Night Guide to C++"
Chapter 7 pg 180 - 184
Chapter 8 pg 192 - 207
MORE ABOUT FUNCTIONS
MORE ABOUT TYPES

Administrative...

- Assignment #4 graded
  - Average Score: ##
  - There were ## assignments turned in
- Assignment #5 due Thursday
- Prelim #1 on 3/15
  - That's 2 weeks from Thursday
  - during class, closed book

Overloading the Assignment Operator

- In lectures past, we've talked about copy constructors
  - Called when a new object is created and set equal to an existing instance
- What's the difference between the following lines of code?

```
MyString str1,str2;
…
// What's the difference between the following two
// lines of code?
MyString aString = str1;
str2 = str1;
```

- The first assignment involves a copy constructor since a new object is being created.
- The second is straight assignment to an existing variable, so no copy constructor is involved.

Overloading the Assignment Operator (cont)

- Remember, C++ will define a default and naïve copy constructor for you if you don't provide one.
- It will just copy member variables (potential for dangling pointers)
- In the case of MyString, we'd need to override the default copy constructor to make sure the storage was copied properly.

```
MyString::MyString(MyString &aCopy)
{
  // Copy storage into new instance if necessary...
}
```

- Again, this will take care of the case where someone tries to assign to a `MyString` variable when it is declared:
  - `MyString aStr = anotherStr;`

Overloading the Assignment Operator (cont)

- However, when we need to handle the case where an existing variable is assigned a new value via the assignment operator, we *overload* the assignment operator:
- The **=** operator is another special case binary operator...

```
MyString &MyString::operator=(const MyString &sourceStr)
{
  // Again, we're cheating a bit by using an existing
  // member function, but it works!  Remember that the
  // unary + operator is overloaded to return a C++ string
  setValue(+sourceStr);
  return *this;   // Huh?
}
```

- Remember that when overloading the = operator you are going to be assigning to an existing instance. If that instance has dynamically allocated data it should be freed.
- We return a reference so that `str1 = str2 = str3` works...

Overloading the Assignment Operator (cont)

- Oh, yeah… we should always make sure that our source and destinations aren't the same..
- We do this by adding the following code:

```
MyString &MyString::operator=(const MyString &sourceStr)
{
  // Make sure we actually have two pointers
  if (this != &sourceStr)
    setValue(+sourceStr);
  return *this;   // Huh?
}
```

- What is "`this`", anyway?
- This is a pointer to the current instance of the class we are in.

Demonstration #1

`MyString`:
Overriding the Assignment Operator

More About Types

- Do you know how to write a type name?
- There is a simple convention for writing a type name…
- Start with a variable declaration of the desired type, then remove the variable...

```
int k;      // Type is really just "int"
int *k;     // Type is really just (int *)
int k[];    // Type is really just (int [])
int *k[];   // Type is really just (int *)[]
int (*k)[] // Type is really just (int []) *
```

- Remember, the asterisk binds tighter than the square brackets
- Another way to "define" our own user types is through the `typedef` keyword.
- This is a way of creating more of a "shorthand" for existing types rather than actually defining a new type.

typedef

- A typedef allows to create a new name for a more complex type.
- The general format of the statement is
  - *typedef <type> <typeName>*
- Consider how we might typedef a pointer to integer:

```
typedef int *intPtr;
main()
{
  intPtr iPtr = new int();
}
```

- After the `typedef` we can use `intPtr` as a "built in" type.
- Notice we don't need to use an asterisk to denote that `iPtr` is a pointer.
- It's built right into the type definition

typedef (cont)

- When using typedef to define a shorthand for some array type, place the right brackets just to the right of the name chosen for the new type.
- Consider a new type called `String255` which is an array of 255 characters (well, plus 1 to account for the NULL byte)

```
// Define a type to represent C style strings of 255
// characters (or less). Leave an extra byte for the NULL
// terminating byte.

typedef char String255[256];
```

- Again, this defines a new type named `String255` which is an array of 256 characters.
- You may also use previously `typedef`'d types in other `typedef` statements...

typedef (cont)

- Consider a new type named `StringArray` which defines an array of `Str255` types:
- It could either be defined as a pointer or as an array itself

```
// Define a type to represent C style strings of 255
// characters (or less). Leave an extra byte for the NULL
// terminating byte.
typedef String255 *StringArray;      // arbitrary size
typedef String255 StringArray15[15]; // 15 String255's
```

- OK, let's take a look at some of this in action...

Demonstration #2

`typedef`

Type Equivalence

- If two types are equivalent they can be assigned to each other without needing to have a specially overloaded assignment operator.
- Two types are equivalent if they have the same name
  - Remember, typedefs don't define new types, just provide shortcuts

```
typedef Student *StudentPtr;
typedef Student *UndergradPtr;
// Both StudentPtr and UndergradPtr are equivalent.
StudentPtr oneStudent;
UndergradPtr anotherStudent = new UndergradPtr();

oneStudent = anotherStudent;  // this is legal because they
                              // are type equivalent
```

sizeof Operator

- The size (in bytes) that any data type takes up may be retrieved by the user by calling the `sizeof` function.
- In C++, this information is really only useful if you are writing an alternative to `new`.

```
int main()
{
  cout << "sizeof(int) is " << sizeof(int) << endl;
  cout << "sizeof(float) is " << sizeof(float) << endl;
  cout << "sizeof(Branch) is " << sizeof(Branch) << endl;

  return 0;
}
```

- For some structures/classes sizeof() might return a value larger than the sum of all fields in question (padding).

Type Conversions

- Early on we touched on the issue of type conversions.
- When assigning between two different types (especially numeric) C++ will do it's best to *implicitly convert* between the type you are assigning from to the type you are assigning to.

```
int main()
{
  int n = -7;
  unsigned int u = n;
  int m = INT_MAX;    // INT_MAX is largest possible int
  float fm = m;
  int pi = 3.142;

  cout << "n = " << n << ", u = " << u << endl <<
          "m = " << m << ", fm = " << fm << endl <<
          "pi = " << pi << endl;
}
```

Demonstration #3

Implicit Type Conversions
(Numeric)

Type Conversions

- What about non-numeric types?
- Well you *can* convert between pointers and integers and between pointers to different types…
- But you need to *typecast* them, like this:

```
int main()
{
  Control *ctrl1 = new PopupMenu(5,5,100,20,"My Menu");
  PopupMenu *pm;

  pm = ctrl1;   // No, the compiler won't let you do this!
  pm = (PopupMenu *) ctrl1;  // But this is ok...
}
```

- A typecast is written as follows:
  - *(typename) expression*

Type Conversions (cont)

- But why does a type cast make it "suddenly legal" to assign between types?
- C++ makes the assumption (perhaps naïvely) that the programmer knows what he or she is doing! :-)
- I could have just as easily (and erroneously) done the following:

```
int main()
{
  Control *ctrl1 = new PopupMenu(5,5,100,20,"My Menu");
  PopupMenu *pm;
  int arbitraryInt = 345345;

  pm = ctrl1;   // No, the compiler won't let you do this!
  pm = (PopupMenu *) arbitraryInt;  // But this is ok ????
  pm->setNumItems(5);               // YIKES!!!!!!!
}
```

Type Conversions (cont)

- Typecasting can be a powerful tool, especially when dealing with derived classes needing to be accessed from a base class pointer.
- Consider the following pseudo-code...

```
// The following is pseudo-code, it is not complete...
int main()
{
  MenuObject *itemList[50]
  itemList[0] = new MenuItem(…); // Assume constructors
  itemList[1] = new SubMenu(…);

  // Now, typecast our way to the derived classes
  ((MenuItem *) itemList[0])->setCmd(…);
  ((SubMenu *) itemList[1])->appendItem(…);
}
```

Type Conversions (cont)

- The moral of the story is to be very, very careful with typecasting
- Essentially, it overrides the compiler's type checking mechanism
- So you can do some pretty bizarre things
- But, used responsibly, you can do useful things as well.
- Did you know that you can define what it means to typecast an instance/reference to a class you've defined?
- Consider the following code...

```
int main()
{
  MyString aStr("This is a test");
  string cppStr = aStr;  // Compiler won't like this!
}
```

Overloading Typecasts

- We could just use the `+aStr` to make the compiler happy, but there's a better way.
- We can overload the (`string`) typecast in `MyString`...

```
MyString::operator string() const
{
  return MakeString();
}
```

- Let's check this one out...

Demonstration #4

Overloading typecasts

Overloading Typecasts (cont)

- It would be very tempting to try and overload more typecasts to make our life easier.
- How about overloading the (int) typecast to return the integer value of the string?
- That would alleviate the need to call MyString::MakeInt().
- We could define it as follows:

```
MyString::operator int() const
{
  return MakeInt();
}
```

Overloading Typecasts (cont)

- There's a problem, though.
- Even though this will work as advertised it has an interesting side effect.
- Consider the following code:

```
int main()
{
  MyString aStr("This is a test");
  cout << "aStr is : " << aStr << endl;
}
```

- When run, this code will print out: `aStr is : 0`
- Why? Because the << operator is called for cout, it sees that aStr has an integer typecast operation and it prefers it to the string typecast we defined earlier. Be careful...

Final Thoughts

- Assignment #5 is due on Thursday
- Assignment #6 will be posted by Thursday
- Come to office hours with problems
- Prelim #1 3/15 in class