Lecture 13

Miscellaneous Topics I

Default Arguments

- Suppose we want to write a new global function that searches for characters in a string.
- We want to be able to search for the first occurrence of a specified character from a specified starting point in the string.
- We might implement is as follows:

```
int findCharInString(string s,char c,int startPos)
{
   // Search for the specified character
  for (int k=startPos; k<s.length(); k++)
    if (s[k] == c)
      return k;  // Found it!  Return the index to the caller

  // Didn't find it, return -1;
  return -1;
}
```

Default Arguments (cont)

- Now suppose we wanted to give the user the flexibility of not having to specify the starting position:
  - In most cases we're probably going to be starting from index 0 anyway
- We could add an additional global function which overloads `findCharInString()`, like this:

```
int findCharInString(string s,char c)
{
  // In this member function, we're just going to call
  // the *real* findCharInString() with 0 as the third arg:
  return findCharInString(s,c,0);
}
```

Default Arguments (cont)

- I would probably place the declarations of findCharInString() in a header file for easy inclusion in code that wants to use it.

```
//
// MyUtilities.h
//

int findCharInString(string s,char c);
int findCharInString(string s,char c,int startPos);
```

Default Arguments (cont)

- What we're really doing with findCharInString() is providing two definitions, one of which uses a *default value* for one of the parameters.
- However instead of going to the trouble of having two functions defined, C++ gives us a way to specify a default value for a parameter right in the declaration (but not the definition):

```
//
// MyUtilities.h
//
int findCharInString(string s,char c,int startPos=0);

// MyUtilities.cpp
#include "MyUtilities.h"
int findCharInString(string s,char c,int startPos)
{ … }
```

Default Arguments (cont)

- To have the compiler use the default value for a given argument I simply omit that argument when calling the function.
- This means that default arguments *must* come at the end of a function declaration.
- In other words, you cannot have an argument with a default value specified appear before a regular argument (with no default specified)

```
int main()
{
  string aStr = "This is a test";

  int pos = findCharInString(aStr,'s');
  cout << "the first s is at position: " << pos << endl;
  cout << "the next on is at " <<
          findCharInString(aStr,'s',pos+1) << endl;
}
```

Demonstration #1

Using Default Arguments

Default Arguments (cont)

- Remember, parameters with default values need to appear at the *end* of your parameter list.
- Once you choose to take the default value when calling a function with default values in it, all subsequent parameters must take the default as well.

```
int findCharInString(string s,char c,int start = 0);
int findCharInString(string s,char c,int start = 0, int stop);  // ???
int findCharInString(string s,char c,int stop, int start = 0);  // ???
int findCharInString(string s,char c,int start = 0, int stop = -1);
```

- Sometimes default arguments can be awkward:
  - No clean way to specify a default value for `stop` (length of string)
  - We could omit the default value, but then we'd need to put `stop` before `start`!
  - We resort to "flag passing" (-1 to mean we want to search until the end of the string)

Default Arguments (cont)

- Let's take a look at how that might be implemented:

```
int findCharInString(string s,char c,int startPos,
                     int stopPos)
{
  // Determine what our stopping point is.  Introducing
  // some really weird notation...
  int stopAt = (stopPos == -1) ? s.length()-1 : stopPos;

  // Search for the specified character
  for (int k=startPos; k<=stopAt; k++)
    if (s[k] == c)
      return k;  // Found it!  Return the index to the caller

  // Didn't find it, return -1;
  return -1;
}
```

Default Arguments (cont)

- According to LNG, a default value "can be any expression, it needn't be a constant. Note, though, that any variables involved are statically bound, so be careful when using default argument values with virtual functions."
- Stroustrup makes no reference to being able to specify a default value with a variable.
- Savitch doesn't really say.
- It would have been useful in our `findCharInString()` function to pass `s.length()` as the default value for the `stopPos` argument, but...
- You cannot use member variables or function calls(unless they are *static*, but we haven't covered static members yet)
- You can use global variables
- Let's look at our `findCharInString` modification.

Demonstration #2

Using Default Arguments
(and the stopPos flag)

void and void *

- A `(void *)` type is a pointer to *anything.*
- That is, you can assign any pointer to a variable of type `(void *)`.
- The reverse is not true however.
- You cannot assign a `(void *)` variable to a pointer variable (except another `(void *)`) without explicit type casting.

```
int main()
{
  char *foo = "This is a test";  // Did you know this is legal?
  void *somePtr;
  somePtr = foo;
  foo = (char *) somePtr;
}
```

- So why would you use a `(void *)` anyway?

void and void * (cont)

- In short, use a (void *) anytime you need to deal with a pointer of any type.
- Usually, this is as a parameter to a function.
- Consider a hex dump function… A low level function to do a hex dump should be able to take any pointer:

```
void hexDump(void *ptr, long size)
{
  char *p = (char *)ptr;
  for (int j=0; j<size; j+=16)
  {
    cout << hex << (unsigned long)p+j << ": ";
    for (int k=j; k<j+16 && k<size; k++)
      cout << hex << (unsigned char) p[k] << " ";
    cout << endl;
  }
}
```

Returning References

- Remember, a reference is *like* a pointer (pointers are used to implement references), so when you return a reference to an object it's like returning a pointer.
- Remember, too, that functions which return pointers usually allocate the memory they return a pointer to. Otherwise the potential for dangling pointers exists.
- Since you can't dynamically allocate memory directly to a reference (like you can for a pointer) you are more likely to return pointers than references when performing this type of work.
- So when *is* it a good idea to return a reference?
- Let's take a brief detour first.

A Simple Array Class

- Consider the following class used to implement an array

```
Class MyArray
{
public:
  MyArray()
  {  for (int k=0; k<50; k++)  internal[k] = 0; }
  int element(int k)
  {  if ((k > 0) && (k<50)) return internal[k];
     return 0;
  }
  int setValue(int k,int val)
  { if ((k > 0) && (k<50)) internal[k] = val;}

private:
  int internal[50];
};
```

A Simple Array Class (cont)

- We could set and get values in this array like this:

```
void main()
{
  MyArray a;
  a.setValue(0,55);
  a.setValue(1,44);
  a.setValue(2,43);
  cout << "Element #1 is: " << a.element(1) << endl;
}
```

Returning References (cont)

- MyArray::element() can be used to access an individual elements.
- So, it's the only way to *get* items from the array.
- BUT, if we change our definition so that it returns a reference...

```
int &MyArray::element(int k)
{
  if ((k >= 0) && (k < 50))
    return internal[k];
}
```

- There is an interesting side effect. We can now put a call to this function on the *left side* of an assignment operator.
- That is, we can now make assignments to a given element in our array using a call to this function...

Demonstration #3

MyArray::element as an l-value

Special Binary Operator Overloading

- I mentioned earlier that binary operators are (in general)overloaded globally, not within a class.
- There is an exception.
- The `[ ]`operator is considered a binary operator (pointer and index)

```
int &MyArray::operator[](int k)
{
  return element(k);
}
```

- So, now we can access characters in our strings like array elements...

Demonstration #4

Overloading []

Lecture 13

Final Thoughts