Const, References, and Templates

Const

We've seen how to define constants in C using the `#define` pre-processor directive. For example:

```
#define MAX 10
```

In C++, we can also define constants with the keyword "`const`". Here's the format:

```
const type name = value;
```

(You don't have to give the constant a value here. However, once you do initialize the constant, you cannot change its value.) Here's an example:

```
const double PI = 3.14159;
```

Then if we tried to change `PI`:

```
PI = 3.14;
```

we would get a compilation error. The `const` keyword can be used on local variables, global variables, and class/instance variables.

Constant parameters

Consider the following function, which is supposed to print every value in an array of integers:

```
void print(int *a, int length) {
     int i;
     for (i = 0; i < length; a[i]++) {
          cout << a[i] << endl;
     }
}
```

What this function actually does is infinitely increment the first element in the array, since we have "`a[i]++`" instead of "`i++`". In fact, we don't want this function to be able to modify the array at all. To force this behavior, we can make the array parameter constant:

```
void print(const int *a, int length) {
     int i;
     for (i = 0; i < length; a[i]++) {
          cout << a[i] << endl;
     }
}
```

Now, we would get a compiler error in this function, since we modified a constant parameter.

It's a good idea to make parameters constant if you don't intend to change them in the function. This basically creates a warning system for you to ensure that you're not changing things by accident.

Constant member functions

You may also want a way to forbid object member function from changing any instance variable. For example, consider the following class definition:

```
class Vector3 {
     private:
          int x, y, z;
     public:
          Vector3(int, int, int);
          void addToEach(int);
          void print(void);
};
```

Here, we probably don't want the `print` function to inadvertently change the values of the vector. Here's how we would change the `print` prototype:

```
class Vector3 {
     private:
          int x, y, z;
     public:
          Vector3(int, int, int);
          void addToEach(int);
          void print(void) const;
};
```

And here's what the corresponding implementation would be:

```
Vector3::Vector3(int x, int y, int z) {
     this->x = x;
     this->y = y;
     this->z = z;
}

void Vector3::addToEach(int num) {
     x+=num;
     y+=num;
     z+=num;
}
```

```
//print cannot change x, y, or z
void Vector3::print(void) const {
     cout<<"("<<x<<", "<<y<<", "<<z<<")"<< endl;
}
```

If we changed the `print` function to modify the values of `x`, `y`, or `z`, we would get a compiler error.

References

A *reference* is very similar to a pointer, but a lot of the details are handled behind the scenes. References in C++ are very similar to how objects are treated in Java – they're both really memory addresses. A reference variable is an alias for an existing variable, and we cannot change it to reference another variable. Here is how a reference is declared:

```
type &name = variable;
```

This establishes `name` to be an alias for `variable`. For example:

```
Vector3 v1(1, 2, 3);
Vector3 &v2 = v1;
```

Now, `v2` is an alias for `v1`. We treat `v2` just like we would an ordinary object, using **.** notation to access members. For example, we could do:

```
v2.addToEach(3);
```

Since `v2` is really an alias for `v1`, `v1` would now have the values (4, 5, 6). However, if we had a situation like this:

```
Vector3 v1(1, 2, 3);
Vector3 v2(4, 5, 6);
v2 = v1;            //v2 now holds the values (1, 2, 3)
                    //but v2 is a different object than v1

v2.addToEach(3);    //v2 = (4, 5, 6)
                    //v1 is still (1, 2, 3)
```

Since `v2` is not a reference, when we set `v2` to be `v1`, we actually copy all the instance variables from `v1` to `v2`. When we call `v2.addToEach(3)`, then `v2` has the values (4, 5, 6). However, `v1` is unchanged.

References and `NULL`

Recall that `NULL` is a constant designated for pointers that have not yet been assigned a value. If a function is passed a pointer, we must first verify that the pointer is not `NULL`.

References, although they are very similar to pointers, cannot be `NULL`. We cannot have a reference variable without making it the alias of another object in memory. Therefore, **it makes no sense to compare a reference to `NULL`** (and in fact it will not compile).

C++ call-by-reference
Recall the call-by-reference `swap` function from C:

```
void swap(int *a, int *b) {
     int temp = *a;
     *a = *b;
     *b = temp;
}
```

We could then swap two numbers like this:

```
int x = 3;
int y = 4;
swap(&x, &y);   //now x = 4 and y = 3
```

A reference is essentially a pointer behind the scenes, so we can write a similar call-by-reference function in C++:

```
void swap(int &a, int &b) {
     int temp = a;
     a = b;
     b = temp;
}
```

And we could call it like this:

```
int x = 3;
int y = 4;
swap(x, y);
```

Since `a` and `b` are reference parameters, they become aliases for the passed variables `x` and `y`. Thus when we change `a` and `b` in the swap function, it actually changes `x` and `y`.

Notice that C++ call-by-reference does not require passing addresses or dereferencing the function parameters. (In fact, this is really what is going on, but all the pointer manipulation is done for you.)

Constant reference parameters
Consider the following function, which uses the `Vector3` class we defined earlier:

```
void printVect(Vector3 v) {
    v.print();
}
```

Now suppose I've created a `Vector3` object and want to call my `printVect` function:

```
Vector3 vect(1, 2, 3);
printVect(vect);
```

When objects are passed in this way (by value), an entirely new object is created and the instance variables are copied over. This means that we create a new `Vector3` object and set its `x`, `y`, and `z` and be the same as `vect`.

This process wastes a lot of time and space. It's a lot more efficient to pass things by reference, where we're essentially passing the address of the object instead of the object itself. However, we don't want the function to change the original object. Solution: pass things by constant reference. Here's how we'd redo the `printVect` function:

```
void printVect(const Vector3& v) {
    v.print();
}
```

We could still call `printVect` in the same manner:

```
Vector3 vect(1, 2, 3);
printVect(vect);
```

Now, we're only passing the address of the object, but we don't have to worry about the function modifying the object.

You should always pass objects by reference (if you want to change the object) **or by constant reference** (if you don't want to change the object).

Function templates

When we implement data structures, we want to be able to hold a variety of types. For example, we might want a linked list of ints, chars, strings, `Vector3` objects, etc. In Java, we can accomplish this by having our data structure hold `Object`s. However, there is no similar base class in C++.

To allow flexible data structures (and flexible functions that can accept more than one type), we need to use something called a *template*. Templates in C++ are very similar to *generics* in Java (generics are a feature of Java 1.5, so you may not have seen them before). This section will discuss how to write functions that accept a variety of types.

Declaring function templates
First, we have to change the prototype for functions that can accept a variety of types. Here's how they'll look:

```
template <class T>
//function prototype
```

Here, `T` is the name of our generic type. We can use `T` as a parameter type or return type in our function. Here's a sample prototype for a function that can compute the minimum of several types of values:

```
template <class T>
T min(T, T);
```

This function accepts two arguments of the same type (whatever type we want), and returns the minimum value.

When we implement a function with a template type, we have to include the "`template <class T>`" line again. Here's the implementation of the min function:

```
template <class T>
T min (T a, T b) {
     if (a < b) return a;
     else return b;
}
```

Calling function templates
When we call a templated function, we must specify which type we're substituting for "`T`". Here's the format:

```
functionName<type>(params);
```

For example, here's how we could compute the minimum of two integers:

```
int num1 = 3;
int num2 = 4;

int result = min<int>(num1, num2);
```

We can similarly compute the minimum of two strings:

```
string s1 = "apple";
string s2 = "banana";

string first = min<string>(s1, s2);
```

(Recall that we can use the < operator with strings to see which one comes first alphabetically.)

How it works
The compiler does nothing with a function template until it sees a call to that function. At that time, the compiler generates a function that replaces the template type with the type passed in. For example, when we computed the minimum of two ints and two strings, the compiler generated the functions:

```
int min(int a, int b) {
     if (a < b) return a;
     else return b;
}

string min(string a, string b) {
     if (a < b) return a;
     else return b;
}
```

Then, the compiler replaces the calls to template functions with calls to the correct "generated" function. For example, it rewrites the min calls as follows:

```
int num1 = 3;
int num2 = 4;

int result = min(num1, num2);

string s1 = "apple";
string s2 = "banana";

string first = min(s1, s2);
```

At this point, template functions work exactly like overloaded functions.

Class templates
We can also use template variables across a class. Here's the format:

```
template <class T>
//class definition, using "T" as a variable type
```

Then, the implementation of each constructor and function that refers to `T` should look like this:

```
template <class T>
//implementation, using ClassName<T> as the scope operator
```

Suppose we want to write a class that holds a pair of elements (of any type). Here’s what the class declaration would look like:

```
template <class T>
class Pair {
     private:
          T first;
          T second;
     public:
          Pair(T, T);
          T max(void);
};
```

And here’s the implementation:

```
template <class T>
Pair<T>::Pair(T first, T second) {
     this->first = first;
     this->second = second;
}

template <class T>
T Pair<T>::max(void) {
     if (first < second) return first;
     return second;
}
```

One caveat: when you write a templated class, the class definition and its implementation must be in the same file. Do not make a separate header file – put both the definition and the implementation in a `.cpp` file.

Now, if we want to create objects with a template class type, we do:

```
className<type> objectName(params);
```

For example, we could create pairs of ints and pairs of strings:

```
Pair<int> intPair(1, 2);
Pair<string> stringPair(“apple”, “banana”);
```

Then, we can use these objects exactly like we’ve done before:

```
int intMax = intPair.max();          //has value 2
string strMax = stringPair.max();    //has value “banana”
```

We can also create pointers to template objects. To do this, use `className<type>` as both the object type and the constructor name. For example, we can create a pointer to a pair of characters:

```
Pair<char> *charPair = new Pair<char>('a', 'b');
```

Then, we can use `charPair` like a normal object pointer:

```
char charMax = charPair->max();
```

Linked list example

The time you'll most often want to use templates with classes is when you're implementing data structures. In this section, we'll look at how to write a linked list that can hold any type you want.

Here's the definition and implementation of the linked list:

```
//linkedlist.cpp
template <class T>
class Node {
	public:
		T data;
		Node<T> *next;
		Node(T d) {data = d; next = NULL;}
};

template <class T>
class LinkedList {
	private:
		Node<T> *head;
	public:
		LinkedList(void) {head = NULL;}
		void add(T);
};

template <class T>
void LinkedList<T>::add(T elem) {
	Node<T> *n = new Node<T>(elem);
	if (head == NULL) {
		head = n;
	}
	else {
		Node<T> *temp = head;
		while (temp->next != NULL) {
			temp = temp->next;
		}
```

```
                temp->next = n;
            }
        }
```

Then, we could write a separate file that used a linked list:

```
        //main.cpp
        #include "linkedlist.cpp"

        int main() {
            LinkedList<int> list;
            list.add(2);
            list.add(3);

            return 0;
        }
```

Standard template library
C++ has many implementations of templated data structures in the *standard template library*. For example, it has the classes:

```
        vector
        list
        deque
        set
        map
        hash_set
        hash_map
```

These are quite similar to the *Java Collection Library*. When you leave this class, it will be preferable to use the standard template library than to write your own data structures. Here is a link to further documentation about each of these classes:

http://www.sgi.com/tech/stl/stl_introduction.html