**C++ Activities and Lecture Notes**

**Chapter 16 - Templates**

We’ve written a lot of routines that sort and print integers. Now we want to repeat the entire exercise with doubles. How do we re-write our code?

Version #1: write overloaded versions and replace “int” with “double.”

So how do we know we got them all? Did we accidently replace an int with a double that should have been left as an int? Do we really want to maintain two versions of the same code?

Version #2: Templates

A template allows a function or class definition to be parameterized by type.

void displayArray(int [] a, int s){

for(int i = 0; i < s; ++i){

cout << a[i] << “ “;

}

cout << endl;

}

becomes

template<typename T>

void displayArray(T [] a, int s){

for(int i = 0; i < s; ++i){

cout << a[i] << “ “;

}

cout << endl;

}

Now we have a displayArray method that works with any data type.

Review: inline functions vs. “standard” functions

Advantages: no function call / return overhead which improves performance, saves room on the stack, increases locality of reference (which helps the cache controller), may be easier to optimize.

Disadvantages: increases code size which may lead to page swapping, makes for a large header file.

Functions defined within the header file are automatically inlined; functions declared in an external .cpp file are (by default) not inline. In an external .cpp file use the inline keyword to signal your desire that a function be handled inline. Note that this is considered a “hint” to the compiler, which will decide whether or not to handle the function as inline or not.

General rule of thumb: functions that are both large and called from multiple locations within a program shouldn’t be inlined. Pretty much everything else should use the inline keyword, and let the compiler decide.

Note that templates are (more or less) the same thing as overloaded functions, but it’s the compiler that does the work. When you invoke a function through a template, the *compiler* writes the appropriate version of the function and inserts it into your code, so templates are actually “function factories” that the compiler uses to write functions specific to the data type used. Therefore, general library functions that use templates must be written in the .h file or defined within the program itself, not in an external .cpp file. Template functions that are methods of a class can be defined in the header file and implemented in the .cpp file. In any event, they’re implemented as inline functions.

**HOMEWORK:**

Alter your existing sort code to use templates. Test it by creating and sorting arrays of

(a) doubles

(b) any objects that we’ve created that support the relational operators

Well, none of our objects do this, do they? Well, last week we learned about overloading, so let’s pick a class and overload the relational operators. OK, let’s pick *two* classes – ComplexNumber and Person.

Comparing complex numbers is actually tricky and there’s good argument for saying that there’s no mathematically legitimate way to do so. Therefore we’ll do lexicographical comparison. For two complex numbers A and B, A < B is Ar < Br, or if Ar == Br , if Ac < Bc. In other words, compare the real parts first. If they’re the same, compare the complex parts.

We’ll also perform lexicographical comparisons for Person / OCCCPerson, in this order:

Last Name

First Name

Birth Date

Student ID Number (OCCCPerson only)

**HOMEWORK:**

Overload the stream and relational operators for ComplexNumber, Person, and OCCCPerson. Write a program that creates arrays of several data types and demonstrate how they can be sorted with the same function call. Note that the overloads in OCCCPerson should take advantage of code already written for Person.