**C++ Activities and Lecture Notes**

**Chapters 17 and 18 – Exceptions and Name Scope Management**

Functions have preconditions and postconditions, or (as some texts say) a contract.

“If you fulfill the preconditions, I will fulfill the postconditions.”

What do you do if the preconditions aren’t met?

(1) Display an error message

This violates cohesion. A method should be focused only on the task at hand. In particular, an arithmetic method has no business even knowing that an output stream exists.

(2) Return an error code

The error code must necessarily be of the same type as the expected data and the calling program may neglect to check for the error code. In this case the error code is used as data, resulting in program failure.

(3) Exit with an error code

If the program can’t continue, this would be appropriate. But what if the error condition could be handled by another part of the program?

(4) Throw an exception

The modern view is – if your preconditions haven’t been met, throw an exception. Whoever catches that exception might be able to correct the issue and allow the program to continue. If not, the program will exit.

**The assert macro**

One way to exit with an error is to use the assert macro. This macro performs a test and exits if the test is false with an error code. For example:

assert(p >= 0);

Asserts are often used for debugging, but are not usually used in production code. They can be disabled by a compiler directive, which means you can leave all of your “debug” asserts in the code, then disable them to compile to a production product.

**Exceptions**

Exceptions in C++ aren’t *too* different from exceptions in Java.

[ write lots of exception demo code ]

**Name Scope Management**

[ scope demo code ]

Three Rules of Program Design:

high coupling

low cohesion

no surprises

Variables with scope larger than necessary, particularly global variables, increase coupling.

Only use them when absolutely necessary. Note that in GUI programming it’s absolutely necessary.

Using variables with the same name that differ only in scope is “surprising.” Don’t do this. Likewise, using variables with very similar names. Your Instructor once had to debug a graphics program that included the following variables, all referring to the current x-coordinate…

x x\_ x\_\_ \_x\_ \_\_x \_xx xx\_

X X\_ X\_\_ \_X\_ \_\_X \_XX XX\_

xX Xx \_xX\_ \_Xx\_ \_x\_x \_xX Xx\_

…and so on. Don’t do this, either.

**Namespaces**

You’re writing a large program, along with several other programmers. You write a “doThis” function, unaware that another programmer, working in a different part of the code, *also* wrote a “doThis” function. Oops!

The problem is that, by default, function names are global. In a large project it’s more appropriate to bundle function definitions into namespaces that are local to the part of the program that uses them.

[ namespace demo code ]

**HOMEWORK**

Let’s revisit one of our very first assignments: Roman Numbers. Rewrite this program to convert between Roman Numbers, decimal, and hexadecimal. The program should be menu-driven, allowing the user to select the desired conversion. Allow both upper- and lower-case letters in the Roman Numbers and hexadecimal values.

Use exceptions to handle the following conditions:

* The user enters an invalid Roman Number (*e.g.* VIX).
* The user enters an invalid hexadecimal number (*e.g.* 12H3).
* The user enters a decimal or hexadecimal value greater than decimal 10,000 (the largest Roman Number we will permit is MMMMMMMMMM).

**STUDY GUIDE**

Chapter 17 Summary and Review Exercises 4, 5, 6, 7, 9, 10, 11, 12, 13, 15, 16, 17

Chapter 18 Summary and Review Exercises 1, 2, 3, 11, 12