Unit 4 Study Guide and Review: Advanced Concepts

4a. Write class and function templates

1. What is a template and what is it used for?

   A template is a generic data type which allows for any data type to be stored in that variable. You use a template when you expect many different data types to be used and creating separate overloaded methods to handle these different versions is unreasonable. A template can be a class or a method.

A template is used to allow a variety of different data types to be used instead of restricting the data type to a single type. Instead of creating a method that only receives an integer, you can create a method that accepts any data type, by using a template. This video explains the purpose and meaning of a template.

2. How do you code a template?

   Example:

```
template <class typeParam>

typeParam max(typeParam a, typeParam b){

    return (a>b ? a:b);

}
```

   You must first declare the class as a template and indicate the template variable name. Then you can define the method name and provide template data types as parameters and set the return type as a template data type.

   These slides provide examples of how to code a template.

4b. Code with a class that manipulates the files

1. What is the difference between character I/O and formatting I/O?

   `getchar()`, `putchar()`, `printf()`, and `scanf()` are all input and output from a keyboard and to the screen. getchar() and `putchar()` deals with single characters, while `printf()` and `scanf()` deal with formatted strings. `getc()`, `putc()`, `fscanf()` and `fprintf()` are all methods that read and write to a file. `getc()` and `putc()` read and write single characters from and to a file. `fscanf()` and `fprintf()` read and write formatted strings from and to a file.

Depending on the purpose of your input and output will determine the methods that you utilize. Whether you use getChar() and putChar(), or printf() and scanf() is dependent on whether you are reading and writing a file, one character at a time or all at once. To explore these options, review this article

2. How do you code input and output?

This video shows a great example of how to write the code for reading from a file. To explore how to write to a file, review this video.

- **ofstream:** Stream class to write on files
- **ifstream:** Stream class to read from files
- **fstream:** Stream class to both read and write from/to files.

Reading a text file is very easy using an ifstream (input file stream).

1. **Include** the necessary headers.
2. `#include <fstream>`
   `using namespace std;`
3. **Declare** an input file stream (ifstream) variable. For example,
4. `ifstream inFile;`
5. **Open** the file stream. Path names in MS Windows use backslashes (\). Because the backslash is also the string escape character, it must be doubled. If the full path is not given, most systems will look in the directory that contains the object program. For example,
6. `inFile.open("C:\\temp\\datafile.txt");`
7. **Check** that the file was opened. For example, the open fails if the file doesn't exist, or if it can't be read because another program is writing it. A failure can be detected with code like that below using the ! (logical not) operator:
8. ```
   if (!inFile) {
      cerr << "Unable to open file datafile.txt";
      exit(1);   // call system to stop
   }
   ```
9. **Read** from the stream in the same way as cin. For example,
10. ```
    while (inFile >> x) {
      sum = sum + x;
    }
    ```
11. **Close** the input stream. Closing is *essential* for output streams to be sure all information has been written to the disk, but is also good practice for input streams to release system resources and make the file available for other programs that might need to write it.
12. `inFile.close();`

Reading from a file can also be performed in the same way that we did with cin:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( getline (myfile,line) )
    {
      cout << line << '\n';
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

4c. Use namespaces and exceptions in C++ code

1. What is a namespace and what is its purpose?

Namespaces are declared regions that define scope. The purpose of a namespace is to organize code into logical groups. You can create your own namespaces, which would then allow you to organize your content into a logical scope. For more information on how to use namespaces, watch this video

A namespace is so that you can have functions named the same things in your local functions that are in a standard library or external library. You just put namespace name {} in the header of the class or function declarations then when using or instantiating use the scope operator::

Namespaces are used to define a scope, and allow us to group global classes, functions, and/or objects into a single group. To use a namespace, you can either put the keyword namespace in front of any method definition and use the `namespace::` notation to refer to the namespace, or you can use the keyword `using namespace` and the name of the namespace:

2. What is an exception and what is its purpose and when is it used?

An exception is an error that is thrown when specific conditions are met that prevent processing to continue. Many classes throw an exception when a problem occurs. An exception allows the processing to continue or allow the program to exit gracefully instead of causing a run-time error. If you run your program and it throws an error, perhaps you should create a try catch that catches this error, while also seeking to find the reason for the error. For further information, review this article.

An exception occurs when a problem in the code or data prevents the program from completing processing normally. By handling the exception, you allow the program to process the exception without a program crash. Many of the predefined classes used in C++ have a predefined error that it throws when a problem exists. If you define this exception and tell the compiler how to handle it, you will allow your program to end gracefully or skip any problem data.

4d. Use recursive functions

1. What is a recursive function and how is it used?

   Recursion is a method of function calling in which a function calls itself during execution. Recursion is easy to code but carries a lot of overhead.

A recursive function is a function that calls itself as many times as necessary until a specific condition is met. It behaves much like a loop in that it repeats itself, but is not usually part of a loop structure. A popular use of the recursive method is the application of the fibonacci number. A fibonacci number is the sum of the two numbers before it: 1 2 3 5 8 13….Each iteration through the loop catures the sum of the previous two numbers and then returns and repeats the process. For more information about recursion, read this page.

4e. Write preprocessor instructions

1. What are the different types of preprocessor functions and when are each type used?

The preprocessors consist of the directives, which give instruction to the compiler to preprocess the information before actual compilation starts. Start all preprocessor directives with a #, which should appear at the top of the page, with only blank lines in front of it. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;). You already have seen a #include directive in all the examples. This macro is used to include a header file into the source file. There are number of preprocessor directives supported by C++ like #include, #define, #if, #else, #line, etc. For a list of preprocessor commands, review this article. Be sure to know the different types and under what conditions they are used.

1)**#pragma**

Each implementation of C and C++ supports some features unique to its host machine or operating system. Some programs, for instance, need to exercise precise control over the memory areas where data is placed or to control the way certain functions receive parameters. The #pragma directives offer a way for each compiler to offer machine- and operating-system-specific features while retaining overall compatibility with the C and C++ languages. Pragmas are machine- or operating-system-specific by definition, and are usually different for every compiler.

2)**#define**

You can use the #define directive to give a meaningful name to a constant in your program.

3)**#error**

When #error directives are encountered, compilation terminates. These directives are most useful for detecting programmer inconsistencies and violation of constraints during preprocessing.

4)**#undef**

As its name implies, the #undef directive removes (undefines) a name previously created with #define.

5)**#if, #elif, #else, #endif**

The #if directive, with the #elif, #else, and #endif directives, controls compilation of portions of a source file. If the expression you write (after the #if) has a nonzero value, the line group immediately following the #if directive is retained in the translation unit.

6)**#include**

The #include directive tells the preprocessor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears.

7)**#ifdef, #ifndef**

The #ifdef and #ifndef directives perform the same task as the #if directive when it is used with defined(identifier).

Unit 4 Vocabulary

This vocabulary list includes terms that might help you with the review items above and some terms you should be familiar with to be successful in completing the final exam for the course.

Try to think of the reason why each term is included.

- Catch block
- Try block
    - "Try" and "**catch**" are keywords that represent the handling of exceptions due to data or coding errors during program execution. A try **block** is the **block** of code in which exceptions occur. A **catch block catches** and handles try **block** exceptions.
- Exception
- Exception handling
    - An exception occurs when a problem in the code or data prevents the program from completing processing normally. By handling the exception, you allow the program to process the exception without a program crash. Many of the predefined classes used in C++ have a predefined error that it throws when a problem exists. If you define this exception and tell the compiler how to handle it, you will allow your program to end gracefully or skip any problem data.
- Namespace
    - Namespaces are declared regions that define scope.
- Pre-processor
    - **Preprocessors** are programs that processes our source code before compilation.
- Raised exception
    - *Raising an exception* is a technique for interrupting the normal flow of execution in a program, signaling that some exceptional circumstance has arisen, and returning directly to an enclosing part of the program that was designated to react to that circumstance.
- Recursion

    - **Recursion** is a process in which a function calls itself as a subroutine. This allows the function to be repeated several times, since it calls itself during its execution.
- Scope
    - Each name that appears in a C++ program is only valid in some possibly discontiguous portion of the source code called its *scope*.
- Standard error
    - Stderr the error output stream (usually file)

- Standard input
    - Stdin the input stream (usually keyboard or file)
- Standard output
    - Stdout the output stream (usually screen or file)
- Streams
    - A 'stream' is internally nothing but a series of characters. The characters may be either normal characters (char) or wide characters (wchar_t). Streams provide you with a universal character-based interface to any type of storage medium (for example, a file), without requiring you to know the details of how to write to the storage medium. Any object that can be written to one type of stream, can be written to all types of streams. In other words, as long as an object has a stream representation, any storage medium can accept objects with that stream representation.
- Template
    - Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.
- Thrown
    - In the event of an exception control of the program is transferred (thrown) to the specified handler for the exception.