Lecture 6: Header Files

Introduction

More advanced programs will require reusing code. Instead of copying and pasting code from one file to another, the reusable code can be stored in a file called a *header file*, a user-defined library that can be imported into other files. They are created by saving the file with the extension .h.

Header Files

A header file is an extended global scope that is not executable. This means all declaration and definition permissions and restrictions for the global scope are the same for header files. Likewise, they need to be included in executable files (.cpp) similar to standard libraries [using the preprocessor directive `#include`] except they are enclosed in double quotation marks to debug them.

Since a header file can be included in many files including other header files, there will be instances when multiple copies of its content will be included in files, which adds unnecessary overhead. Using a *header guard* avoids this issue. It ensures that a single copy of the content of its header file will be included in any file when imported. It uses three preprocessor directives named *#ifndef*, *#define*, and *#endif* that tell the compiler to import the header file if it has not yet been imported. A template of a header file with a header guard is

```
#ifndef id-code
#define id-code
content
#endif
```

where the *id-code* is a name; typically, the name of the header file in all uppercase letters.

Example:
A header file named `test.h` would be written as

```
#ifndef TEST_H
#define TEST_H
//code
#endif
```

Namespaces

With the inclusion of multiple header files, ambiguous naming errors will arise; thus, using *namespaces*, named global scopes, becomes essential. All declaration and definition permissions and restrictions in the global scope are the same for namespaces. Furthermore, a namespace can expand to multiple files and have multiple instances in a single file. Its syntax is

```
namespace identifer{members}
```

where *members* are everything that can be declared or defined in the global scope.

Anyway, namespaces are the same if their identifiers are the same even if they are in the same file or separate files; thus, for an expansive namespace, be careful not to have multiple members with the same identifier in it since an error will occur.

Example: The headers illustrated below, '`file01.h`' and '`file02.h`', are using a single namespace that have several instances

```
#ifndef FILE01_H
#define FILE01_H
namespace exo{int a(int);}
//same namespace
namespace exo
{
  int a(int x) {return x * x + x;}
}
//different namespace
namespace eso {int a;}
#endif
```

```
#ifndef FILE02_H
#define FILE02_H
//same namespace
namespace exo
{
  char t(int ch)
  {
    return ((ch >= 0)?((char)('a'+ (ch % 26))):('0'));
  }
}
#endif
```

Whenever a member of a namespace needs to be accessed outside of the scope of the namespace, the *scope resolution operator* (`::`) is used. Its syntax is

namespace`::`*member*

Additionally, to place a member or all members of a namespace into a scope (omit the scope resolution operator), use the keyword *using*. As a directive, the using keyword can bring a member into a scope. Its syntax is

`using` *namespace*`::`*member*`;`

Furthermore, as a declaration, it can bring the entire namespace into a scope. Its syntax is

`using namespace` *namespace*`;`

Only use the declaration if it is known that no ambiguous naming error will occur. Moreover, placing the directive or declaration in the global scope after including the libraries makes the members visible in every scope.

Example: The code segment refers to the '`file01.h`' and '`file02.h`' from the previous example.

```
#include <iostream>
#include "file01.h"
#include "file02.h"
using exo::t; //t is in every scope

int main()
{
  using namespace std; //std members in main scope
  eso::a = 78; //access int variable a
  cout << exo::a(6) << eso::a; //access both as, output: 4278
  cout << t(26); //output 'a'
  return 0;
}
```