COS10007 - Developing Technical Software

Week 9

Modular software decomposition

&

Features of C++ that improve on C as a procedural language

Dr Prince Kurumthodathu Surendran

Single Source File

- Most programs are built from more than one source
- Doing everything in one source file is unorganized - It's like keeping all your books in one big box eventually the first book get lost at the bottom.
- It is better to organise the files in multiple sources.

Multiple Source Files

- Advantages to use multiple source files :
  - with each source containing a "module" of functionality, usually self-contained so that it can be programmed separately and tested separately.
  - write different parts of your program in different languages.
  - reduces debugging cycle, because most of the sources will already be compiled when you go to rebuild your program after fixing that one last bug.

C Header Files

- A header file is a file with extension .h which contains C function declarations and macro definitions and to be shared between several source files.

- There are two types of header files:
  1. the files that come with your compiler, e.g. stdio.h
  2. the files that the programmer writes

- You request the use of a header file in your program by including it, with the C preprocessing directive #include like you have seen inclusion of stdio.h header file, which comes along with your compiler.

Header file vs. Copying all contents

- Including a header file is equal to copying the content of the header file, but this was not done because
  - it will be error-prone
  - it is not a good idea to copy the content of header file in the source files, specially if when there are multiple source files comprising the program.

- A simple practice in C or C++ programs is to keep all the constants, macros, system wide global variables, and function prototypes in header files and include that header file wherever it is required.

Global Symbol

- Symbols, in this context, are functions and variables.
- Global symbols are symbols that are known to the whole program. For example:

```
int c;
int fun(int x)
{
int y = x * 2 + 1 + c;
printf("fun(%d)=%d\n", x, y);
}
```

- There are two global symbols: `c` and `fun`

Global Symbol

- The symbols `c` and `fun` are global because:
  - `c` is a variable declared outside any functions,
  - `fun` is a function that other parts of the program will be able to use.
- Note that, `printf` is a global symbol too.
- Even though the code does not define `printf`, it can be used here.
- The difference is that `fun` is defined by programmer but `printf` is external.
- The header of stdio.h is usually included in order to use the global symbol of `printf` – the prototype of `printf` function (name, return type and parameters) reside in stdio.h.

Include Syntax

- To include a header file, the preprocessing directive `#include` is used.

- It has following two forms:
  1. `#include <myfile>`
  2. `#include “myfile"`

- First form is used for system header files. It searches for a file named `myfile` in a standard list of system directories.
- Second form is used for header files of your own program. It searches for a file named `myfile` in the directory containing the current file/program.

Include Operation - example

- For example, if you have a header file myheader.h as follows:

```
#define MYNAME “Nobadi”
```

- and a main program called program.c that uses the header file, like this:

```
int x;
#include "myheader.h"
int main ()
{
      printf (“My Name is: %s.”, MYNAME);
}
```

Include Operation - example

- the compiler will see the same token stream as it would if program.c read

```
int x;
#define MYNAME “Nobadi”
int main ()
{
      printf (“My Name is: %s.”, MYNAME);
}
```

Include Operation

- The `#include` directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file.

- The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the `#include` directive.

Example - myfile.h

```
int add(int a, int b);
```

Example - myfile.c

```
#include"myfile.h"

int add(int a, int b)
{
      return(a + b);
}
```

Example - myprog.c

```
#include<stdio.h>
#include"myfile.h"

int main() {
    int num1 = 10, num2 = 10, num3;
    num3 = add(num1, num2);
    printf("Addition of Two numbers : %d", num3);
    return 0;
}
```

How to compile?

```
gcc –o a myprog.c myfile.c
```

* All files of myProg.c and myfile.c and myfile.h must be in the same folder.

Once-Only Headers

- If a header file happens to be included twice, the compiler will process its contents twice and will result an error.
- To prevent this is, enclose the entire real contents of the file in a condition, for example:

```
#ifndef HEADER_FILE
#define HEADER_FILE
/* the entire header file*/
#endif
```

- When the header is included again, the conditional will be false, because `HEADER_FILE` is defined.
- The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Example - other.h

```
#ifndef _OTHER_H_
#define _OTHER_H_
int luckyNumber();
#endif
```

Example - other.c

```
#include "other.h"
int luckyNumber()
{
      return 7;
}
```

Example - main.c

```
#include <stdio.h>
#include "other.h"
int main ()
{
   printf("%d\n", luckyNumber());
   return 0;
}
```

Selective Includes

- Sometimes it is necessary to select one of several different header files to be included into a program.
- For example, they can specify configuration parameters to be used on different operating systems.
- Do this with a series of conditions as follows:

```
#if SYSTEM_1
#include "system_1.h"
#elif SYSTEM_2
#include "system_2.h"
#elif SYSTEM_3
...
#endif
```

- The preprocessor offers the ability to use a macro for the header name.

- Instead of writing a header name as the direct argument of #include, you simply put a macro name instead, e.g.

```
#define SYSTEM_H "system_1.h"
...
#include SYSTEM_H
```

- SYSTEM_H will be expanded, and the preprocessor will look for system_1.h as if the #include had been written that way originally.

Summary

- Programs that are long and complex should be decomposed into multiple source files, each with a name ending in .c (for C programs)
- Group functions that manipulate the same data structures or have related purposes into the same file.
- All types, functions, global variables, and manifest constants that are needed by more than one source file should also be declared in a header file a .h file.
- Don't declare function bodies (or anything that causes the compiler to generate code or allocate space) in the header file (except for inline functions).
- Each source file should refer to those header files it needs with an #include line.
- Never #include a .c file.

Introduction to C++

- C++ was developed by Bjarne Stroustrup at Bell Labs and was originally called “C with classes”.

- C++ improves on many of C's features and provides object-oriented-programming (OOP) capabilities that increase software productivity, quality and reusability.

OOP

- OOP is a programming paradigm with
  - classes,
  - objects,
  - encapsulation,
  - overloading,
  - Inheritance, and
  - Polymorphism

- Apart from that, C++ also enable generic programming (with function templates and class templates).

Compilation

- Like C, C++ programs are directly compiled into native code by a compiler – programs compiled into native code at compile time tend to be quicker than those translated at run time, due to the overhead of the translation process.

Adding two numbers

```
#include <iostream> // allows program to perform input and output

int main()
{
   int number1; // declare an integer, 1st to add

   std::cout << "Enter first integer: "; // prompt user for data
   std::cin >> number1; // read 1st integer from user into number1

   int number2; // 2nd integer to add
   int sum;

   std::cout << "Enter second integer: ";
   std::cin >> number2;
   sum = number1 + number2; // add the numbers; store result in sum
   std::cout << "Sum is " << sum << std::endl; // display sum; end line
} // end function main
```

Output

```
Enter  first  integer:  45
Enter second integer: 72
Sum is 117
```

C++ (1)

- C++ filenames can have the extensions of .cpp, .cxx or .C (uppercase) .
- Begin a comment with // and use the remainder of the line as comment text (for single line).
- The input/output stream header file <iostream> (from the standard library) must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output.
- The .h header file have been replaced by the C++ Standard Library header files.
- Declarations can be placed almost anywhere before the corresponding variables are used in the program (can be immediately before).

C++ (2)

- The standard output stream object (std::cout) and the stream insertion operator(<<) are used to display text on the screen.
- The standard input stream object (std::cin) and the stream extraction operator (>>) are used to obtain values from the keyboard.
- The stream manipulator std::endl (means end line) outputs a newline, then flushes the output buffer.
- The notation std::cin, std::cout, std::endl specifies that we're using a name, that belongs to namespace std.

C++ (3)

- This is required when we use standard C++ header files.
- Using multiple stream insertion operators(<<) in a single statement means it is concatenating, chaining or cascading stream insertion operations.
- In C++, if program execution reaches the end of main without encountering a return statement, it's assumed that the program terminated successfully
- Therefore it is safe to omit the `return 0;` statement at the end of main in C++ programs.

Similarities between C and C++

- /* */ also can be used for multiple lines comment.
- Same as C, program begins execution with function main.
- Keyword int to the left of main indicates that main "returns" an integer value . If not returning, use void.
- In C++, a parameter list with empty parentheses is equivalent to specifying a void parameter list.
- Calculations can also be performed in output statements, e.g.:
  - `std::cout << "Sum is " << number1 + number2 << std::endl;`

C++ Standard Library

- C++ is rich with collections of existing classes and functions in the C++ Standard Library.
- It is important to learn how to use the classes and functions in the C++ Standard Library.
- Use existing functions wherever possible, i.e. reuse, this practice is one of the core concept of object oriented programming.

C++ Standard Library – advantages

- The advantages of using C++ Standard Library functions and classes are, e.g.
  - improve program performance, because they are written to perform efficiently
  - shortens program development time
  - improves program portability, because they are included in every C++ implementation.

Header file

- The C++ Standard Library is organized in different header files.
- The header files contain:
  - the function prototypes for the related functions that form each portion of the library
  - definitions of various class types and functions
  - constants needed by those functions.
- A header file facilitate the compiler with library and user-written components.
- A programmer-defined header files can be created as .h, and included with the #include preprocessor directive.

C++ standard library header files

Headers	Meaning
`<iostream>`	Contains function prototypes for the C++ standard input and standard output functions.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data.
`<string>`	Contains the definition of class string from the C++ Standard Library.
`<limits>`	Contains classes for defining the numerical data type limits on each computer platform.

C++ standard library header files

Headers	Meaning
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation) , and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<cstring>`	Contains function prototypes for C-style string-processing functions.
`<cmath>`	Contains function prototypes for math library functions.
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions.
`<ctime>`	Contains function prototypes and types for manipulating the time and date.

Calculate the volume of a cube

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

inline double cube( const double side ) {
   return side * side * side; // calculate the cube of side
} // end function cube

int main() {
   double sideValue; // stores value entered by user

   for ( int i = 1; i <= 3; i++ ) {
      cout << "\nEnter the side length of your cube: ";
      cin >> sideValue; // read value from user

      // calculate cube of sideValue and display result
      cout << "Volume of cube with side "
         << sideValue << " is " << cube( sideValue ) << endl;
   }
} // end main
```

Inline Functions

- Function calls involve execution-time overhead.
- C++ provides inline functions to reduce function call overhead-especially for small functions.
- Using inline before a function's return type in the function definition allows the compiler to generate a copy of the function's code in place to avoid a function call.
- This can reduce execution time but may increase program size.
- Therefore, the inline qualifier should be used only with small, frequently used functions.

Example explained

- Definition of inline function cube.
- Definition of function appears before function is called, so a function prototype is not required.
- First line of function definition acts as the prototype.
- Keyword **`const`** in the parameter list of function **`cube`** tells the compiler that the function does not modify variable **`side`**.
  - This ensures that the value of side is not changed by the function when the calculation is performed.
- **`using`** statements helps eliminate the repeating use of **`std::`** prefix.

Output

```
Enter the side length of your cube: 1.0
Volume of cube with side 1 is 1

Enter the side length of your cube: 2.3
Volume of cube with side 2.3 is 12.167

Enter the side length of your cube: 5.4
Volume of cube with side 5.4 is 157.464
```

Keywords for both C and C++

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

Keywords for C++

and	and_eq	asm	bitand	bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	false
friend	inline	mutable	namespace	new
not	not_eq	operator	or	or_eq
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

Boolean

- C++ also provides type `bool` for representing boolean (true/false) values.
- The two possible values of a `bool` are the keywords `true` and `false`.
- When `true` and `false` are converted to integers, they become the values 1 and 0, respectively.
- When non-boolean values are converted to type bool, non-zero values become true, and zero or null pointer values become false.

Pass arguments to functions

- Two ways to do that:
  - pass-by-value
  - pass-by-reference

- Both have its own advantages and disadvantages

Call-by-value

- The argument's value is copied and passed to the called function.
- Advantage:
  - Changes to the copy do not affect the original variable's value in the caller.
  - This prevents the accidental side effects that affect the system reliability and correctness.
- Disadvantage:
  - if a large data item is being passed, copying the data will take longer execution time and huge amount of memory.

Call by reference

- Advantage:
  - Better performance, because it can eliminate the pass-by-value overhead of copying large amounts of data.

- Disadvantage:
  - Weaken security; the called function can corrupt the caller's data.

Reference Parameters

- It is an alias for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference,
  - in the *function prototype*: add an ampersand (&) after the parameter's type, e.g. `int &`
  - in the *function header*: use the same notation when listing the parameter's type, e.g. `int &a`
  - in the *function call*, simply mention the variable by name to pass it by reference
  - in the *function body*, simply mention the variable by its parameter name

Pass arguments to functions (1)

```
#include <iostream>
using namespace std;

// function prototype (value pass)
int squareByValue( int );

// function prototype (reference pass)
void squareByReference( int & );

int main()
{
   int x = 2; // value to square using squareByValue
   int z = 4; // value to square using squareByReference

   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue( x ) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;

   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;
} // end main
```

Pass arguments to functions (2)

```
// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue( int number )
{
   return number *= number;
      // caller's argument not modified
} // end function squareByValue



// squareByReference multiplies numberRef by itself and
// stores the result in the variable to which numberRef
// refers in function main
void squareByReference( int &numberRef )
{
   numberRef *= numberRef; // caller's argument modified
} // end function squareByReference
```

Constant Reference Parameters

- For passing large objects efficiently, use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object.
- The called function will not be able to modify the object in the caller.
- Use `const` before the type, e.g. `const double side`.
- Modifiable arguments are passed to functions by using pointers, small non-modifiable arguments are passed by value and large non-modifiable arguments be passed by using references to constants

Thank You