CMPT 115: Principles of Computer Science

Lab 4: Const, Multiple File Compilation, ADTs, and file redirection

Department of Computer Science
University of Saskatchewan

February 1 – 5, 2016

Laboratory 4 Overview

Part 0 Pre-lab Reading

1. A short bit on `const`
2. Function Prototypes
3. Header Files and Multiple File Compilation
4. ADTs as multi-file C++ programs
5. A short tutorial on File redirection

Hand in : Exercises (to hand in by the end of the week)

- Exercise 1 (page 31)
  File(s): `lab4exercise1.cc`
- Exercise 2 (page 32)
  File(s): `testTime.cc myTime.h myTime.cc`

Part I

Pre-Lab 3 Reading

Constants

- Data that does not change can be stored in variables.
- Use the `const` keyword to indicate a variable whose initial value cannot change ever.
- A `const` variable must be initialized in its declaration.
- Any attempt to change a `constant` variable will be flagged by the compiler as an **error**.
- Use this idea to
  - Give names to values; names are more meaningful anyway, e.g. `pi`
  - Prevent accidental modification of important values, e.g. `pi`
- You can store constants as global `const` variables, e.g. `pi`

Constants: Examples

```
const double PI = 3.141592653589793;

const char* str = "Hello";          // make str immutable

int max(const int a, const int b) {
  // function is not allowed to change a or b!
  if (a>b) { return a; }
  return b;
}
```

- Using `const` keeps the programmer honest!
- The `const` tell the compiler to flag an error if anyone tries to change a const variable.
- This simple picture gets considerably more complex, but we'll leave it here for now.

C++ Function definitions and prototypes

Function Definition

```
float average(float a[], int b){
   float sum = 0;
   for (int i = 0; i < b; i++)
     sum = sum + a[i];
   return sum/b;
}
```

- Provides an *implementation*, i.e., the body of the function.

Function Prototype

```
float average(float [], int);
```

- Describes the *interface* to a function, but has no implementation.
- The parameter names can be omitted.

The purpose of function prototypes in C++

- Prototypes are used to describe how functions can be used, without giving the complete body.
- Three purposes:
  1. To assist the compiler when a program is separated into several files.
  2. To assist the compiler when two functions call each other.
  3. To assist the compiler when the programmer wants to define functions in an order that makes sense to human readers.

  Examples of each will follow!

When 2 C++ functions call each other

In the following example, even() calls odd(), and odd() calls even().

```
bool even(int n) {
  if (n < 0) return even(-1*n)
  else if (n == 0) return true;
  else if (n == 1) return false;
  else return odd(n-1);
}
bool odd(int n) {
  if (n < 0) return odd(-1*n)
  else if (n == 0) return false;
  else if (n == 1) return true;
  else return even(n-1);
}
```

If even() is defined first, the compiler complains that odd() is not defined, and vice versa. (some languages, e.g., Java, don't have this problem) In C/C++, a function must be defined before it is used.

When 2 C++ functions call each other

Adding a prototype gives the compiler enough information to work with to solve the problem.

```
bool odd(int);

bool even(int n) {
  if (n < 0) return even(-1*n)
  else if (n == 0) return true;
  else if (n == 1) return false;
  else return odd(n-1);
}
bool odd(int n) {
  if (n < 0) return odd(-1*n)
  else if (n == 0) return false;
  else if (n == 1) return true;
  else return even(n-1);
}
```

Define your functions in any order

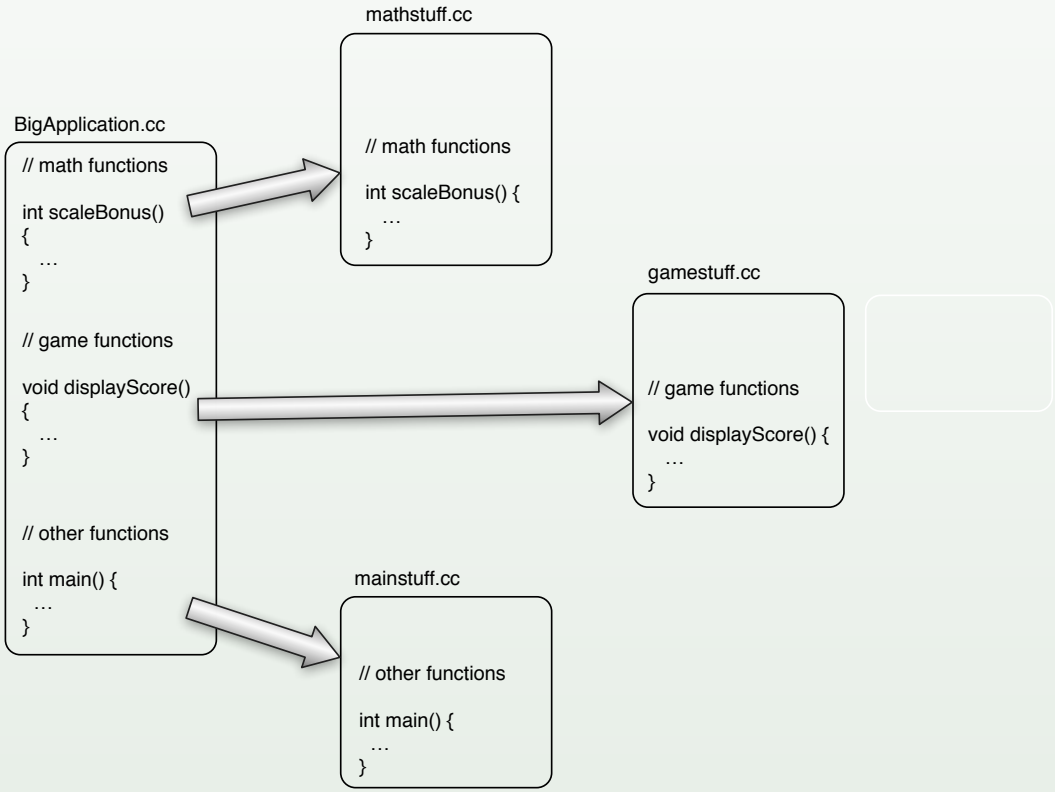
If you put all function prototypes before any function definitions, the function definitions can go in any order.

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

// prototypes first
float *myFunc(float *, float *);
void anotherFunc(int);

// followed by function definitions in any order
int main() {
...
}

void anotherFunc(int a){
...
}
float *myFunc(float *x, float *y){
...
}
```

Example: Splitting a large program into separate files

To manage large programs, it is a good idea to separate the program into several files

- Each file should contain functions that are related to each other
- Each file has a nice name giving some hint about its contents



Large Programs: Example

- Example: Suppose you separated your program's functions into three files: `mathstuff.cc`, `gamestuff.cc`, and `mainstuff.cc`.
  You compile your program as follows:

```
g++ -o runprog mathstuff.cc gamestuff.cc
                    mainstuff.cc
```

- Notes:
  - Order of the files is not important.
  - Exactly one of these files must contain a `main()`.
- **Important Note:** If function A in `gamestuff.cc` calls function B in `mathstuff.cc`, then a prototype for function B **must** appear in `gamestuff.cc`.

Large Programs: Function Protypes needed!

- **Important Note:** If function A in `gamestuff.cc` calls function B in `mathstuff.cc`, then a prototype for function B **must** appear in `gamestuff.cc`.
- This is most easily done by creating "header files", which contain function prototypes.
- Create a **header file** for every program file, containing function prototypes of functions defined in the program file.
- Example:
  1. Create a new file `gamestuff.h` to contain the function prototypes of functions defined in `gamestuff.cc`
  2. Use the directive `#include "gamestuff.h"` in `mathstuff.cc`.
  3. Do the same for `mathstuff.cc` and `mainstuff.cc`

Example: Create headers for each .cc file

BigApplication.cc

```
// math functions

int scaleBonus()
{
  ...
}

// game functions

void displayScore()
{
  ...
}


// other functions

int main() {
  ...
}
```

mathstuff.cc

```
// math functions

int scaleBonus() {
  ...
}
```

mathstuff.h

```
// math prototypes

int scaleBonus();
```

gamestuff.cc

```
// game functions

void displayScore() {
  ...
}
```

gamestuff.h

```
// game prototypes

void displayScore();
```

mainstuff.cc

```
// other functions

int main() {
  ...
}
```

mainstuff.h

```
// other prototypes
```

Example: Include headers in other .cc files

BigApplication.cc

```
// math functions

int scaleBonus()
{
  …
}

// game functions

void displayScore()
{
  …
}


// other functions

int main() {
  …
}
```

mathstuff.cc

```
#include "mathstuff.h"
#include "gamestuff.h"
#include "mainstuff.h"

  // math functions

  int scaleBonus() {
    …
  }
```

mathstuff.h

```
// math prototypes

int scaleBonus();
```

gamestuff.cc

```
#include "gamestuff.h"
#include "mathstuff.h"
#include "mainstuff.h"

 // game functions

 void displayScore() {
   …
 }
```

gamestuff.h

```
// game prototypes

void displayScore();
```

mainstuff.cc

```
#include "mainstuff.h"
#include "gamestuff.h"
#include "mathstuff.h"

  // other functions

  int main() {
    …
  }
```

mainstuff.h

```
// other prototypes
```

Example: End result

mathstuff.h

```
// math prototypes

int scaleBonus();
```

mainstuff.h

```
// other prototypes
```

No prototype for main()

gamestuff.h

```
// game prototypes

void displayScore();
```

mathstuff.cc

```
#include "mathstuff.h"
#include "gamestuff.h"
#include "mainstuff.h"

  // math functions

  int scaleBonus() {
    …
  }
```

mainstuff.cc

```
#include "mainstuff.h"
#include "gamestuff.h"
#include "mathstuff.h"

  // other functions

  int main() {
    …
  }
```

gamestuff.cc

```
#include "gamestuff.h"
#include "mathstuff.h"
#include "mainstuff.h"

  // game functions

  void displayScore() {
    …
  }
```

You compile your program as follows:

```
g++ -o runprog mathstuff.cc gamestuff.cc mainstuff.cc
```

Including header Files

- We include *header files* to inform the compiler about functions implemented somewhere else.
- We have been doing this for a long time: `<iostream>`, `<cstdlib>` `<cmath>`, and others.
- These define function prototypes!
- We use angle-brackets to inform the compiler to find these files in a standard location (i.e., not in your working directory)
- Header files which you have written yourself are included using double quotes instead, e.g., `#include "myheader.h"`.
- The double quotes force C++ to look for these files in your working directory.

More about header files

Header files, or "`.h` files"

Header file contain the relevant information necessary to be able to **use** the functions/data structures written.

- Defined global constants (`const`).
- Function prototypes.
- Definition of any `structs` needed by the prototypes.

Source files, or "`.cc` files"

Source files contain all the implementation of all the functions in the header files.

- Any `#include` statements that are needed by the implementations, **including the source file's own header file!**
- Function definitions (implementations).

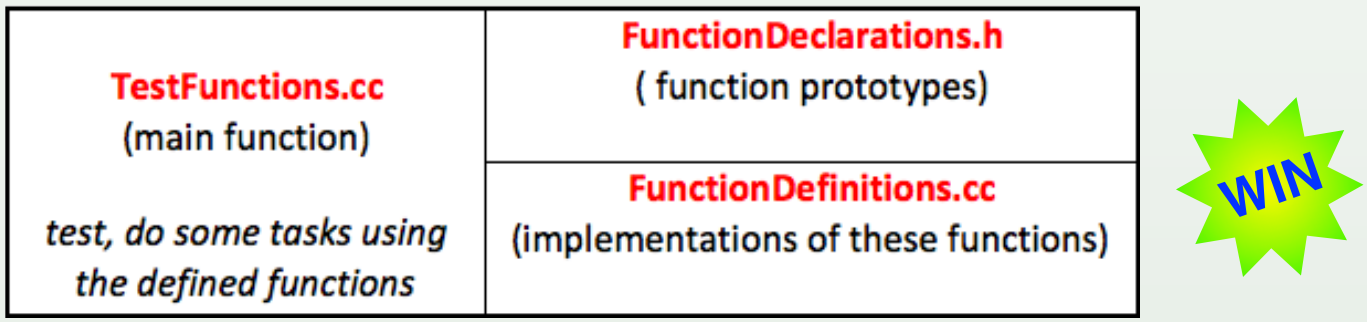
Recap: Compilation of Multiple Files

It is a good idea to write a program solely for testing your functions.

- Example: suppose `testmath.cc` has a main() function that only tests functions from `mathstuff.cc`. Then you could compile:

```
g++ -o testmath mathstuff.cc testmath.cc
```

- This lets you test each set of functions without having to write a complete application, or cluttering up your application's main() function!



Review - ADTs

Recall from lecture notes:

Abstract Data Type

An abstract data type consists of:

- Encapsulation of the data.
- Interface of operations on the data.
- Implementation of the operations.

ADTs using multifile compilation

- We will implement our ADTs in a program source file: e.g, `ADT.cc`
- The interface will be made available through a header file: e.g, `ADT.h`
- An application source file may `#include "ADT.h"` but not `ADT.cc` itself.
- An application source file must only interact with the ADT by the operations declared in `ADT.h`.

Example: Point ADT layout

point.h

```
struct Point {
    int x;
    int y;
};


// Algorithm reflectInXAxis(pt)
// Pre: pt :: refToPoint
// Post: changes the point referred to by 'pt'
//       to the same point reflected in the x axis.
void reflectInXAxis(Point *pt);


// Algorithm moveForwardByAngle(pt, angle, distance)
// Pre: pt :: refToPoint
//      angle :: float, between 0 and 360,
//      distance :: integer, positive
// Post: changes the point  'pt' by moving it forward
//       a distance 'distance', at an angle of 'angle'
void moveForwardByAngle(Point *pt,
                        float angle,
                        float distance);
```

point.cc

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

// include the interface
#include "point.h"

// Algorithm reflectInXAxis(pt)
// Pre: pt :: refToPoint
// Post: changes the point referred to by 'pt'
// to the same point reflected in the x axis.
void reflectInXAxis(Point *pt){
  (*pt).x *= -1;
}

// Algorithm moveForwardByAngle(pt, angle, distance)
// Pre: pt :: refToPoint
//      angle :: float, between 0 and 360,
//      distance :: integer, positive
// Post: changes the point  'pt' by moving it forward
//       a distance 'distance', at an angle of 'angle'
void moveForwardByAngle(Point *pt,
                        float angle,
                        float distance)
{
  ... // a much more complex piece of code
}
```

Review: ADT layout

- Make sure there is an algorithm header (pre-,post, and return, as comments) for each function prototype in the `ADT.h` file.
- This way, one can look at the comment and the prototype to know what the function does.
- We only need to examine the `ADT.h` file to use it.
- An application that uses the ADT is written in a different file altogether.
- For example, we might write a main function which uses this ADT, and store it in a file called `ptedit.cc`.

Using UNIX to send data to our programs

- So far, we have always taken data into our programs from the console (cin).
- In this lab, we'll see that we can get data into our programs from a text document (a file) with the help of the UNIX command line.
- We call this "file redirection".
- This is not the only way to get data from a file; it is simply the easiest to start with.
- From the perspective of your program, nothing is new; we use (cin) as normal.
- From the perspective of running your program on the command-line, there is something new to learn.
- File redirection is only available on the command-line; it is not available in Eclipse!
- File redirection only works properly with text documents. Files created with Microsoft Word, or Excel, will probably break you program.

How it works

- On the command line, we can tell Unix to connect `cin` to a text document you have created and saved.
- Once connected by Unix, every `cin` command will take data from the document, instead of waiting for you to type it at the keyboard.
- The data is taken from the document in the order it appears in the document.
- Your program doesn't know it's been connected to document; it simply gets the data Unix sends it.
- Normally, Unix connects `cin` to your keyboard ("standard input"); using file redirection, Unix connects standard input to a document.

Example Program: redirect.cc

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

int main() {
  int i1, i2, i3, i4;
  char c;
  char s[20];
  float f;

  cin >> i1 >> f >> i2 >> s >> i3 >> c >> i4;
  cout << "The integers were: "
       << i1 << " " << i2 << " " << i3 << " " << i4  << endl
       << "The float number was: " << f << endl
       << "The character was: " << c << endl
       << "The string was: " << s << endl;

  return 0;
}
```

Example: text document with data

Suppose you had a document named `input.txt` with the following data saved.

```
12 3.3 8 hello 50 c 70
```

- Each value separated by spaces; one space is the same as many spaces.
- You could put data on separate lines, too.
- Important: the order of the data matches the order of `cin` in the program.
- Important: This document has to be in the same directory as your program!

Example: The command line

Compile your program the normal way:

```
g++ -Wall -pedantic -o myProg redirection.cc
```

Now run your program on the command line, but connect it to the file `input.txt`:

```
./myProg < input.txt
```

The "less-than" symbol $<$ is the UNIX file-redirection operator.
In English this says: "Run the application `myProg` taking input from the file `input.txt`"
The output would be:

```
The integers were: 12 8 50 70
The float number was: 3.3
The character was: c
The string was: hello
```

Notes and Applications

- Be sure to use $<$ not $>$. The $>$ means "send the console output to a file named ..." which could delete any data you already have in that file.
- With file redirection, you can automate a lot of testing, if your testing requires user input.
- Put the user input in a text file once, and you can zip through a lot of tests without retyping any of it on the console!

Part II

Lab activities and exercises

Exercise 1

The program `example1.cc` has the following functions:

- `absoluteValue` - takes a `float` as parameter and calculates and returns the absolute value
- `outputAbsoluteValue` - takes a `float` as parameter and output the absolute value to the screen in a user friendly way.
- `main` - asks the user for a value, and then output the absolute value of that number.

Activity:

1. Try to compile this program as given. Notice the errors!
2. Add function prototypes to this program so that it will compile (without changing the order of the function definitions).
3. Experiment with changing the order of the function definitions. Any order should work if the prototypes are before the definitions.
4. Hand in your `example1.cc` file.

Exercise 2

Activity

- Find the files: `myTime.h`, `myTime.cc` and `testTime.cc` on Moodle.
- Copy them to a folder/directory (a new one would be good)
- The file `myTime.h`
  - defines a record type called `myTime`, that stores time using a 24 clock (the hour is from 0 to 23, and the minutes are from 0 to 59).
  - declares several operations on myTime records
- The file `testTime.cc`
  - Includes `myTime.h`
  - Calls several operations declared in `myTime.h`.
  - Never manipulates any myTime record directly, ever.
- Compile the code by typing

  g++ -Wall -pedantic -o exercise testTime.cc myTime.cc

Exercise 2 continued

ACTIVITY:

1. Add a function to display times on the console (see below)
2. Add a function to add one minute to a given time (see below)
3. Test and demonstrate both new functions by adding code to `testTime.cc`
4. Hand in all your modified files: `myTime.h`, `myTime.cc` and `testTime.cc`

You'll have to modify the interface (add the function prototypes), and the implementation (define the new functions), and call the functions in `testTime.cc`.

`void printTime(myTime *t)`	display the time to the screen in a friendly format.
`void addMinute(myTime *t)`	add one minute to the given time.

Trying out file redirection

ACTIVITY:

1. Download `redirection.cc` and `input.txt` from Moodle.
2. Compile the program normally:

   ```
   g++ -Wall -pedantic -o myProg redirection.cc
   ```

3. Run the program, connecting it to the file using file redirection:

   ```
   ./myProg < input.txt
   ```

4. Open input.txt, and change the text. Rerun your program on the new data!
5. The idea that you can connect programs and files in UNIX is just the start of the study of UNIX. UNIX is a system written for programmers by programmers, with hundreds of useful tools!