CSI 32
Chapter 4: Computation

Today we will talk about

- Functions
- `vector` class

Functions: Why bother with functions?

- We define a function when we want to separate a computation because it
  - makes the computation logically separate
  - makes the program text clearer (by naming the computation)
  - can be used in more than one place in our program
  - eases testing, distribution of labor, and maintenance

Functions: declarations and definitions

- Before a name can be used in C++ program, it must be declared
- We don’t always need to see the definition of the function we are going to use, hence it makes sense to see only *its name, its return type, and its list of parameters*.

Functions: declarations and definitions

- Before a name can be used in C++ program, it must be declared
- We don't always need to see the definition of the function we are going to use, hence it makes sense to see only *its name, its return type, and its list of parameters*.

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

void main()
{
    int x = 10, y = 9;
    cout << max(x,y) << endl;
}

int max(int a, int b)
{
    if (a > b) { return a; }
    else {return b; }
}
```

Functions: declarations and definitions

- A *declaration* is a statement that introduces a name into a scope
  - specifying a type for what is named (e.g. a variable or a function)
  - optionally, specifying an initializer (e.g. initializer value, or a function body)

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

void main()
{
    int x = 10, y = 9;
    cout << max(x,y) << endl;
}

int max(int a, int b)
{
    if (a > b) { return a; }
    else {return b; }
}
```

Functions: declarations and definitions

- A declaration that also fully specifies the entity declared is called a *definition*
- Every *definition* (by definition) is also a *declaration*.

```
int max(int a, int b)
{
    if (a > b) { return a; }
    else {return b; }
}

void main()
{
    int x = 10, y = 9;
    cout << max(x,y) << endl;
}
```

Functions: declarations and definitions

- Some other terminology:

declaration ≈ *prototype*

Header files

- How do we manage declarations and definitions?
- After all, they have to be consistent, and in real-world problems there can be tens of thousands of declarations.

Header files

- How do we manage declarations and definitions?
- After all, they have to be consistent, and in real-world problems there can be tens of thousands of declarations.
- The key to managing declarations of facilities defined “elsewhere” in C++ is the *header*.

- A header is a collection of declarations, typically defined in a file

Header files

- How do we manage declarations and definitions?
- After all, they have to be consistent, and in real-world problems there can be tens of thousands of declarations.
- The key to managing declarations of facilities defined “elsewhere” in C++ is the *header*.

- A header is a collection of declarations, typically defined in a file
- .h is the most common for C++ headers
- Such headers are then #included in our source files

Header files

- How do we manage declarations and definitions?
- After all, they have to be consistent, and in real-world problems there can be tens of thousands of declarations.
- The key to managing declarations of facilities defined “elsewhere” in C++ is the *header*.

- A header is a collection of declarations, typically defined in a file
- .h is the most common for C++ headers
- Such headers are then #included in our source files
- .cpp is the most common for C++ source files

Header files

AHeaderFile.h:

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

testing.cpp:

```
#include “AheaderFile.h”

void main()
{
    int x = 10, y = 9;
    cout << max(x,y) << endl;
}
```

defs.cpp:

```
#include “AheaderFile.h”

int max(int a, int b)
{
    if (a > b) { return a; }
     else {return b; }
}
```

Scope

- A *scope* is a region of program text
- A name is declared in a scope and valid (it is called "*is in scope*") from the point of its declaration until the end of the scope *in which it was declared*

Scope

```
void f()
{
   g();       // error: g() isn't yet in scope
}

void g()
{
   f();       // OK: f() is in scope
}

void h()
{
   int x = y;     // error: y isn't yet in scope
   int y = x;     // OK: x is in scope
   g();           // OK: g() is in scope
}
```

Scope

- We will talk more about the scope later on in course

In-class work

- Create three files: my.h, my.cpp and use.cpp
- The header file my.h contains

```
void print_foo();
int do1(int, int, int);
double do2(int, int, int);
```

- The source file my.cpp `#include`s my.h, iostream, and defines `print_foo()` to print the word “WELCOME”;

  defines `do1` to return the largest of three integer values provided as arguments, and

  defines `do2` to return the average of three integer values
- The source file use.cpp `#include`s my.h, iostream, and defines `main()` find the largest of the three integers taken from a user, as well as their average.

Data for Iteration - Vector

- To do just about anything of interest, we need a collection of data to work on. We can store this data in a `vector`.

For example:

```
// read some temperatures into a vector:
int main()
{
      vector<double> temps;  // declare a vector of type double to store temperatures
      double temp;                    // a variable for a single temperature value
      while (cin>>temp)          // cin reads a value and stores it in temp
                temps.push_back(temp);  // store the value of temp in the vector
      // … do something …
}
// cin>>temp  will return true until we reach the end of file or encounter
// something that isn't a double: like the word "end"
```

Data for Iteration - Vector

- See the program vectorWork.cpp on our webpage

In-class work

- Write a program that reads in a sequence of decimal values from keyboard (from the user), and then displays:
  - the mean (average),
  - the median value, and
  - the largest and the smallest values in the sequence