Coding with your instructor

Please download the source file from the following link:

https://www.cs.uky.edu/~yipike/CS215/testVector.cpp

Try to understand the purpose of functions named dothis() and dothat().

2D vectors

It is also possible to define and use two-dimensional vectors.

- New rows can be added easily.
- And each row can be a different length.
- Really: a vector of vectors.
  `vector<vector<double> > scores`
  - Important: You should have a space in > >
  - Otherwise older C++ versions think you're using the extraction operator.
- Each row of scores is a vector of doubles.

Initializing 2D vectors

2D vectors are a bit of a pain to initialize. We need a nested loop, and a temporary vector for the "current" row.

```
vector<vector<double> > scores; // 0 x 0
for (int i=0; i < STUDENTS; i++)
{
    vector<double> tmpvec;
    for (int j=0; i < HOMEWORKS; j++)
    {
        tmpvec.push_back(100.0);
    }
    scores.push_back(tmpvec);
}
```


Using 2D vectors

Looping over a 2D vector is similar to a 2D array. The biggest difference is that we have to check the size of each row.

```
for (int i = 0; i < scores.size(); i++)
{
    // Process the ith row.
    for (int j = 0; j < scores[i].size(); j++)
    {
        // Process the jth column of the ith row.
        cout << "\t" << scores[i][j];
    }
    // Start a new line after every row.
    cout << endl;
}
```

Introduction to Program Design, Abstraction and Problem Solving

Chapter 7 Header files; streams and I/O

Textbook: C++ for Everyone by Cay Horstmann

CS 215

This time

- Prototypes and header files.
- Reading a line at a time.
- File streams.
- Detecting end of file.
- Streams and functions.
- Command line arguments.

Declaring functions

- C++ must know about the function *before* you can call it.
- A function **prototype** declares the return type, name, and parameters of a function.
  - It doesn't actually define the function: no body.
  - Just add semicolon: `void printMessage(string msg);`
- A **definition** creates something, a **declaration** states that it exists.
  - When you declare a variable, that (usually) also defines it.
  - Function definitions both define and declare the function.
  - Prototypes declare the function but do not define it.

Header files

Usually your declarations go at the top of the file, but sometimes you need to use them in multiple locations.

- A **header file** lets you put declarations in a single place.
  - What goes in a header file?
    - Prototypes of functions
    - Definitions of constants
    - Declarations of global variables
    - Definitions of classes
  - You can create your own header files.
    - Usually end with `.h`
    - Visual studio: right-click "Header Files", Add, New Item, select "Header File" and type a name.

Using header files

- To use a header file in your program, use the `#include` directive.
- System header files use angle brackets, user header files use double quotes:
  - `#include <iostream>`
  - `#include "mathtable.h"`
  - Remember the `.h` !
- It is as though you pasted the contents of the header file instead of writing the `#include` directive.

Double Inclusion Example

grandfather.h:

```
const double PI = 3.1416;

void func1(int n);
```

father.h:

```
#include "grandfather.h"

void func2(double price);
```

child.cpp:

```
#include "grandfather.h"
#include "father.h"
```

Include guards

Including the same header twice could be a problem because of redefined variables and so on. But header files often include one another, so sometimes it is unavoidable.

- Avoid problems with an **include guard (macro guard)**.
  - At the beginning of your header:
    `#ifndef HEADERNAME_H`
    `#define HEADERNAME_H`
  - At the end of your header, `#endif`
  - The **guard macro** `HEADERNAME_H` could be any identifier.
    - ★ It should reflect the name of the header file.
    - ★ All-caps by convention.
  - `#ifndef` stands for "if not defined"
    - ★ Skip everything until the `#endif` if the guard macro is defined.
    - ★ The first time, the macro wasn't defined: define the macro and do stuff.
    - ★ The second time, the macro was defined: do nothing.
    - ★ No more redefinition errors!

Use of Include Guard

```
grandfather.h:
#ifndef GRANDFATHER_H
#define GRANDFATHER_H
const double PI = 3.1416;
void func1(int n);
#endif  /* GRANDFATHER_H */
```

```
father.h:

#include "grandfather.h"

void func2(double price);
```

```
child.cpp:

#include "grandfather.h"
#include "father.h"
```

Use of Include Guard

grandfather.h:

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H
const double PI = 3.1416;
void func1(int n);
#endif  /* GRANDFATHER_H */
```

father.h:

```
#ifndef  FATHER_H
#define  FATHER_H
#include "grandfather.h"
void func2(double price);
#endif    /* FATHER_H */
```

child.cpp:

```
#include "grandfather.h"
#include "father.h"
```

Coding with your instructor

Please download the source file from the following link:

https://www.cs.uky.edu/~yipike/CS215/DemoArraysKey.cpp

We will practice how to define our own header file.

Streams

- You have seen two streams so far: `cin` and `cout`.
- A stream is how C++ represents something you can read input from or write output to.
  - The keyboard and console window, a file (chapter 7), some hardware devices.
  - Usually either standard input/output, or a file.

Input streams

- What happens when you do `cin >> intvar;` ?
  - Read input from the keyboard, but what if the user hasn't typed anything?
  - Wait for something to be typed, but how much?
  - A whole line, but we only need one integer.
    - ★ What is a line? Everything up to a **newline** character `'\n'`
    - ★ Input a line from the user.
    - ★ From the data we have read, skip over the whitespace.
    - ★ Then read characters until we have a complete number.
    - ★ Convert that into an integer and store it in `intvar`.
    - ★ Is anything left over? Yes, at least the newline character.
    - ★ The leftover data is **buffered** (temporarily saved).
    - ★ The next input operation will process the leftover data, and read another line if necessary.

String input

- Reading a single word is straightforward:
  - `string word;`
    `cin >> word;`
  - Skips whitespace, reads a number of non-whitespace characters.
  - Stops just before the next whitespace.
- Lets read a whole line—everything up to the newline.
  - `string line;`
    `getline(cin, line);`

  it reads the newline character, but throws it away.

More about `getline`

- What happens if we do an extraction then a getline?
  - `getline` will read the rest of the line.
  - That might be (probably is) just a newline character!
  - Look at an example.
  - So what to do?

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

int main()
{
      int age;
      string name;

      cout << "Enter your age: ";
      cin >> age;
      cout << "Enter your name: ";
      getline(cin, name);

      cout << endl;
      cout << name << " is " << age
            << " years old." << endl;
      return 0;
}
```

More about `getline`

- What happens if we do an extraction then a getline?
  - `getline` will read the rest of the line.
  - That might be (probably is) just a newline character!
  - Look at an example.
  - So what to do? Discard the rest of the line, then getline.
    - ★ `cin.ignore(256, '\n');` – discard input until 256 characters or a newline character is read, whichever comes first.
- Note that getline does read the newline character.
  - Don't need `cin.ignore` between getlines!

File streams

- C++ defines three **classes** for doing I/O on files.
  - `ifstream` does **i**nput from a plain text **f**ile.
  - `ofstream` does **o**utput to a plain text **f**ile.
  - `fstream` does both input and output on a text or binary **f**ile.
  - All three require `#include <fstream>`
- Before you can use a **file stream**, it must be associated with a file.
  - We do this with the open method:
    ```
    ofstream logfile;
    logfile.open("myprog.log");
    ```
  - It's also possible to open a file when we define it:
    ```
    ofstream logfile("myprog.log");
    ```


Filename parameters

For the compiler which is based on C++98 or earlier version, when opening a file, the filename must be passed as a **C string**, *not* a `string` object. C++11 allows filename to be passed as a string object.

- C strings are character arrays—we'll cover them in Chapter 9.
- For now: a string literal works:
  ```
  logfile.open("myprog");
  ```
- But string variables must be converted with the `c_str()` method:(C++98)
  ```
  string filename = username + ".txt";
  logfile.open(filename.c_str());
  ```
- You can use string variable directly if your compiler supports C++11:
  ```
  string filename = username + ".txt";
  logfile.open(filename);
  ```

File paths

- Where does `open` look for the file? The **current directory (folder)**.
  - In Visual Studio this is usually the solution folder.
  - If you run the executable directly, it is usually the executable's folder.
- You can open files in other directories by using an **absolute path**:
  `out_file.open("C:\\Documents\\cs215\\Downloads\\new.txt");`
- You can also use **relative paths** within the current directory:
  `infile.open("data\\test.txt");`
- Note the doubled backslashes: backslash is special in string literals!

Errors

Opening a file might fail! Why?

- File doesn't exist
  - Only an error for input files.
  - Opening an output file that doesn't exist creates the file!
- Insufficient permissions.
- Full disk.
- Lost network connection.

If there was an error, `open` sets the stream's fail state.

Checking for success

You should *always* check whether the file opened successfully. There are two equivalent ways:

- Check whether either failbit or badbit is set:

```
ifstream logfile("myprog.log");
if (logfile.fail())
    cout << "Error opening log file" << endl;
```

- Or, check whether state of stream is good.

```
ofstream logfile("myprog.log");
if (!logfile.good())
    cout << "Error opening log file" << endl;
```

- Note the opposite logic!
  - `fail()` returns true if the open failed.
  - `good()` returns true if the open succeeded.
- Also: `if (!logfile)`

Closing file streams

- As long as a file stream remains open, you can continue to read from or write to it.
- To finish using a file, finalize any changes to it, and allow other programs to use it, **close** the stream.
  - This happens automatically when the stream goes out of scope.
  - Or close it explicitly with `myfile.close();`
  - It is not possible to do I/O on a closed stream.
    - ★ But you can re-open it on a new (or the same) file.

Reading from files

- Reading from a file works the same as reading from `cin`.

```
string word, name;
infile >> word;
infile.ignore(256, '\n');
getline(infile, name);
```

- But if there is an error it doesn't make sense to prompt for new input: the file isn't going to listen!
  - Instead we can just stop processing when there is an error.
  - You probably want to inform the user, or even exit the program.

Errors and end of file

- We have already seen that the extraction operation detects erroneous input.
  - This puts the stream in a **failing state**: detect with `stream.fail()`
  - Cannot do input from a failing stream: reset with `stream.clear()`
- There is one more another condition to check for: **end of file**.
  - Reading the last line or last piece of input works normally.
  - Then the next attempt to read sets the EOF state.
    - Like the fail state, prevents further reads.
    - Not an error! Every file ends sometime.
  - Check for the EOF state with `stream.eof()`
  - `while (infile >> value)` checks both `fail` and `eof`.
    - Likewise `if (infile.good())` and `if (infile)`

Reach the end of the file, then

If you have reached the end of file on the stream, and you want to go back to the beginning of the file, but the file stream WILL NOT automatically reset its position to the beginning.

- Call `clear()` function to clear the end of file bit first.
- The function `seekg()` allows you to seek to an arbitrary position in a file.
- It sets the position of the next character to be extracted from the ifstream.

```
infile.clear()
// Seeks to the very beginning of the file
infile.seekg(0, infile.beg);
// Seeks to the current poistion of the file
infile.seekg(0, infile.cur);
// Seeks to the end of the file
infile.seekg(0, infile.end);
```


Example: set the position in a file

```
int main()
{
    string line;
    ifstream myFile("test.txt");
    cout << myFile.eof() << endl; // false!
    while (!myFile.eof())
        getline(myFile, line);
    cout << myFile.eof() << endl; // true!
    // Clear the end-of-file bit first
    infile.clear()
    // Seeks to the very beginning of the file
    infile.seekg(0, infile.beg);
    cout << myFile.eof() << endl; // false!
    infile.close();
}
```

Streams and functions

The I/O functions we have written so far use `cin` and `cout`. We can also write functions which use file streams.

- If the function is responsible for reading or writing the whole file, you can open the file within the function.
- But what about functions that handle just part of the file?
  - Could use global variables, but that's bad!
  - Have the function take the stream as a parameter!

```
double read_score(ifstream &infile)
{
    int points, maxpoints;
    infile >> points >> maxpoints;
    return 100.0 * points / maxpoints;
}
```

  - Note that it's passed by reference. Why?
    - ★ Because streams can't be copied.
    - ★ They're always passed by reference.

A function for all streams

- The function we wrote works only on files, not `cin`. Why?

```
double read_score(ifstream &infile)
```

- `cin` isn't an input *file* stream.
- But it is an input stream. One small change:

```
double read_score(istream &infile)
```

  - Every `ifstream` is also an `istream`, so works with this function.
  - And `cin` is also an `istream`, so it works too.
- Note that you can't use file-specific methods like `open` and `close` on an `istream` argument!
  - If you need to, you must use `ifstream`.
- Output is the same: use `ofstream &` or `ostream &`.