CS11 Intro C++

Spring 2018 – Lecture 3

C++ File I/O

- We have already seen C++ stream I/O

```
#include <iostream>
cout << "What is your name?  ";
cin >> name;
cout << "Hello " << name << "!\n";
```

  - Supports primitive types and some C++ classes (e.g. **`std::string`**)
- C++ also includes support for reading and writing files
  - Uses the exact same stream I/O mechanism
  - Simply uses another kind of stream object to read and write
  - **`#include <fstream>`**
- Provides these classes:
  - **`ifstream`** for reading from files
  - **`ofstream`** for writing to files
  - **`fstream`** for reading and writing to files

C++ File I/O (2)

- Usage is very straightforward:

```
vector<double> read_data(string filename) {
    ifstream ifs{filename};
    vector<double> data;

    // Make sure the file was opened successfully
    if (!ifs.is_open())
        throw illegal_argument("Couldn't open file");

    // Read data until we hit EOF
    while (ifs.good()) {
        double v;
        ifs >> v;
        data.push_back(v);
    }
    return data;
}
```

C++ File I/O (3)

- File streams (and all streams) can be used in conditional expressions
  - They will indicate their current status – true for “everything is good!”, or false for “something went wrong!”

```
vector<double> read_data(string filename) {
    ifstream ifs{filename};
    vector<double> data;

    // Make sure the file was opened successfully
    if (!ifs)
        throw illegal_argument("Couldn't open file");

    // Read data until we hit EOF
    while (ifs) {
        double v;
        ifs >> v;
        data.push_back(v);
    }
    return data;
}
```

C++ File I/O (4)

- Note: We don't close the file anywhere in this function!
  - The **`ifstream`** destructor will automatically close the file, when the **`ifs`** object goes out of scope

```
vector<double> read_data(string filename) {
    ifstream ifs{filename};
    vector<double> data;

    // Make sure the file was opened successfully
    if (!ifs)
        throw illegal_argument("Couldn't open file");

    // Read data until we hit EOF
    while (ifs) {
        double v;
        ifs >> v;
        data.push_back(v);
    }
    return data;
}
```

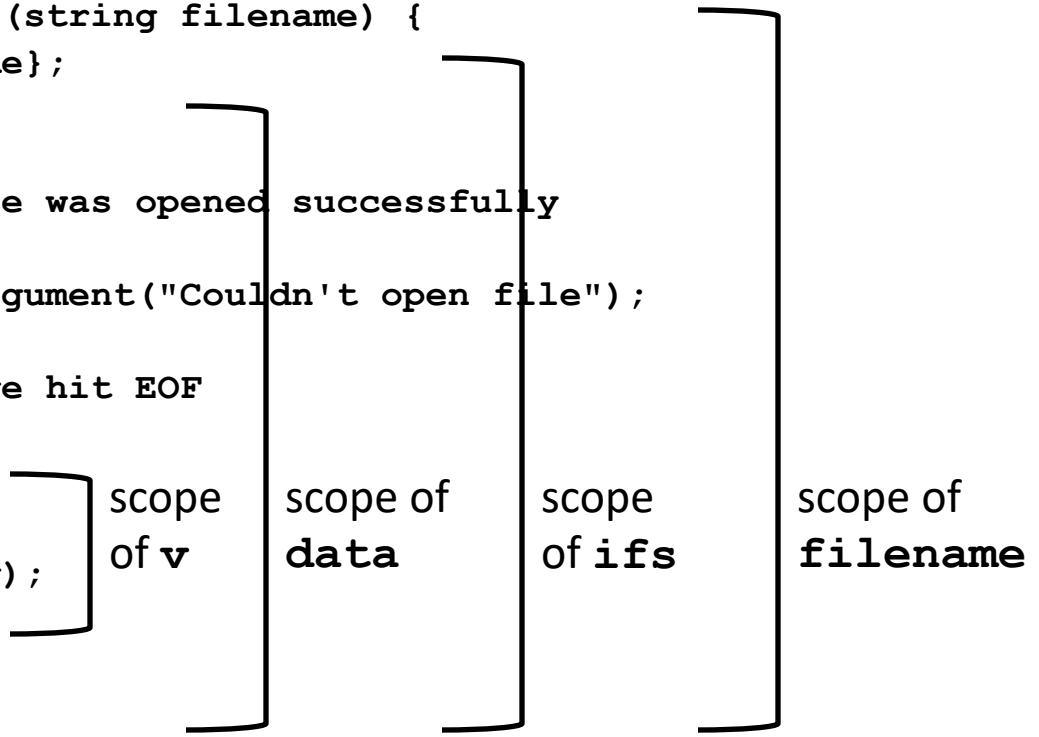
Variable Scope

- A variable's **scope** is the part of the program where the variable is accessible
  - Starts at the variable's declaration; extends to the end of the most immediately enclosing block

```
vector<double> read_data(string filename) {
    ifstream ifs{filename};
    vector<double> data;

    // Make sure the file was opened successfully
    if (!ifs)
        throw illegal_argument("Couldn't open file");

    // Read data until we hit EOF
    while (ifs) {
        double v;
        ifs >> v;
        data.push_back(v);
    }
    return data;
}
```



Variable Scope (2)

- Generally want a variable's scope to be as small as possible
  - Declare variables when and where you actually need them
  - Helps to reduce chances of weird bugs, name conflicts, etc.

```
vector<double> read_data(string filename) {
    ifstream ifs{filename};
    vector<double> data;

    // Make sure the file was opened successfully
    if (!ifs)
        throw illegal_argument("Couldn't open file");

    // Read data until we hit EOF
    while (ifs) {
        double v;
        ifs >> v;
        data.push_back(v);
    }
    return data;
}
```

Variable Scope (3)

- When an object variable goes out of scope, its destructor is called automatically – can perform any necessary cleanup tasks
  - e.g. **`ifstream`** destructor closes the underlying file when **`ifs`** goes out of scope

```
vector<double> read_data(string filename) {
    ifstream ifs{filename};
    vector<double> data;

    // Make sure the file was opened successfully
    if (!ifs)
        throw illegal_argument("Couldn't open file");

    // Read data until we hit EOF
    while (ifs) {
        double v;
        ifs >> v;
        data.push_back(v);
    }
    return data;
}
```

Variable Scope (4)

- Primitive types don't have a destructor
  - e.g. **`int`**, **`double`**, **`float`**, **`long`**, pointer types
  - When **`v`** goes out of scope, its space is reclaimed, but nothing else happens

```
vector<double> read_data(string filename) {
    ifstream ifs{filename};
    vector<double> data;

    // Make sure the file was opened successfully
    if (!ifs)
        throw illegal_argument("Couldn't open file");

    // Read data until we hit EOF
    while (ifs) {
        double v;
        ifs >> v;
        data.push_back(v);
    }
    return data;
}
```

C++ File I/O (5)

- If we wanted to close the **`ifstream`** before it goes out of scope, can use the **`close()`** member-function

- We will cover more details of file I/O in the future…
- Fortunately, basic usage is very straightforward, and uses our existing stream-I/O knowledge!

C++ Function Arguments

- In C++, arguments are **passed by value** as a default
  - The function receives a copy of the arguments, rather than the original
- Example:

```
double compute_distance(Point a, Point b) {
    double dx = b.get_x() - a.get_x();
    double dy = b.get_y() - a.get_y();
    return sqrt(dx * dx + dy * dy);
}

Point p1{3, 5};
Point p2{8, 6};
cout << compute_distance(p1, p2);
```

- **`compute_distance()`** receives a copy of **`p1`** and **`p2`**, rather than the original variables **`p1`** and **`p2`** themselves

C++ Function Arguments (2)

- Passing large objects by value can get very expensive…

```
double compute_average(vector<double> values) {
    double sum = 0;
    for (double v : values)
        sum += v;
    return sum / (double) values.size();
}
```

- If our collection holds 10 million values, copying the data will be *very slow*…

- C++ also supports **passing arguments by reference**, when pass-by-value is undesirable
  - No copy is made!
  - Rather, the function operates on the exact object passed in by the caller

C++ Function Arguments (3)

- Updated function, passing the vector by reference:

```
double compute_average(vector<double> &values) {
    double sum = 0;
    for (double v : values)
        sum += v;
    return sum / (double) values.size();
}
```

  - Now our function receives a **reference** to the caller's vector object, rather than a **copy** of the vector

- References have the exact same syntax as objects
  - The only change we have made to our function was to pass by reference, rather than pass by value

C++ Function Arguments (4)

- Updated function, passing the vector by reference:

```
double compute_average(vector<double> &values) {
    double sum = 0;
    for (double v : values)
        sum += v;
    return sum / (double) values.size();
}

vector<double> input_data;
... // Load input data
cout << compute_average(input_data);
```

- Don’t need to explicitly convert an object into a reference before invoking the function – it happens automatically

C++ Function Arguments (5)

- Passing an object by-reference allows a function to change the caller's object
- Sometimes this is desirable, e.g.

```
void load_input_data(vector<double> &values,
                     string filename) {
    ifstream ifs(filename);
    while (ifs) {
        double v;
        ifs >> v;
        values.push_back(v);
    }
}

vector<double> input_data;
load_input_data(input_data, "data.txt");
cout << compute_average(input_data);
```

C++ Function Arguments (6)

- We definitely don't want **`compute_average()`** to mutate its argument!

```
double compute_average(vector<double> &values) {
    double sum = 0;
    for (double v : values)
        sum += v;
    return sum / (double) values.size();
}

vector<double> input_data;
load_input_data(input_data, "data.txt");
cout << compute_average(input_data);
```

- Can specify that argument's value cannot change by using the **`const`** modifier:

```
double compute_average(const vector<double> &values) {
```

C++ Function Arguments (7)

- When using **`const`**, a function's declaration and definition must match
  - A value's **`const`**-ness is part of the value's type

- In header (**`.h`**) file (or earlier in the **`.cpp`** file):

```
double compute_average(const vector<double> &values);
```

- In source (**`.cpp`**) file:

```
double compute_average(const vector<double> &values) {
    double sum = 0;
    for (double v : values)
        sum += v;
    return sum / (double) values.size();
}
```

Guidelines for C++ Argument Passing

When passing an object as an argument to a function:

- If the function should not modify the object at all, you should pass it by **`const`** reference
  - Avoids the overhead of making a copy of the object
  - Avoids the risk of the function accidentally introducing side-effects
  - *This is the most common scenario!*
- If the function is supposed to mutate the object on behalf of the caller, pass it by non-**`const`** reference
  - Allows the function to mutate the actual object passed by the caller
  - Tends to be a very uncommon situation
- If the function implementation wants to mutate the argument, without those changes being visible to the caller, pass by value
  - The function will receive a copy of the argument, which it can change to its heart's content, without the caller seeing the changes
  - Will incur copying overhead
  - Also tends to be uncommon, but can be very useful technique

Guidelines for C++ Argument Passing (2)

When passing a primitive as an argument to a function:

- e.g. **`int`**, **`long`**, **`char`**, **`float`**, **`double`**, a pointer type

- These values are small, and fast to pass as arguments – generally will always pass them by value
  - Passing them by reference or **`const`**-reference can actually be slightly *slower* than passing them by value!
- If the function is supposed to mutate the primitive value on behalf of the caller, pass it by non-**`const`** reference
  - Again, allows the function to mutate the actual variable passed by the caller
  - Also tends to be a very uncommon situation

User-Defined Classes and **`const`**

- Our previous example:

```
double compute_distance(Point a, Point b) {
    double dx = b.get_x() - a.get_x();
    double dy = b.get_y() - a.get_y();
    return sqrt(dx * dx + dy * dy);
}
```

- Need to change this to use **`const`** references
  - Function doesn't change its arguments
  - Want to avoid overhead of copying these objects
- Updated code:

```
double compute_distance(const Point &a, const Point &b) {
    double dx = b.get_x() - a.get_x();
    double dy = b.get_y() - a.get_y();
    return sqrt(dx * dx + dy * dy);
}
```

User-Defined Classes and **`const`** (2)

- Updated code:

```
double compute_distance(const Point &a, const Point &b) {
    double dx = b.get_x() - a.get_x();
    double dy = b.get_y() - a.get_y();
    return sqrt(dx * dx + dy * dy);
}
```

- Unfortunately, the compiler won't accept this program ☹
- Issue: The compiler doesn't know that **`get_x()`** and **`get_y()`** do not mutate the **`Point`** objects they are called on
- Need to update our class declaration/definition to indicate that **`get_x()`** and **`get_y()`** don't mutate the object they are called on

Point Class Declaration – **point.h**

```
// A 2D point class
class Point {
    double x, y;                // Data-members

public:
    Point();                    // Constructors
    Point(double x, double y);

    ~Point();                   // Destructor

    double get_x() const;       // Accessors
    double get_y() const;
    void set_x(double x);       // Mutators
    void set_y(double y);
};
```

Point Class Definition – **point.cpp**

```
// Returns X-coordinate of a Point
double Point::get_x() const {
    return x;
}

// Returns Y-coordinate of a Point
double Point::get_y() const {
    return y;
}

// Sets X-coordinate of a Point
void Point::set_x(double x) {
    this->x = x;
}

// Sets Y-coordinate of a Point
void Point::set_y(double y) {
    this->y = y;
}
```

User-Defined Classes and **`const`** (3)

- Updated code:

```
double compute_distance(const Point &a, const Point &b) {
    double dx = b.get_x() - a.get_x();
    double dy = b.get_y() - a.get_y();
    return sqrt(dx * dx + dy * dy);
}
```

- Once our **`Point`** class specifies that **`get_x()`** / **`get_y()`** don't change the object they are called on, this code will compile and work perfectly

This Week's Homework

- Complete the functionality of our units-converter
- Initialize the collection of unit-conversions from a data file, rather than specifying conversions in the code
  - Update main program to report errors when file can't be opened, or when file contents specify a conversion rule more than once
- Update your entire program to pass arguments by references, and use the **`const`** keyword, wherever it is appropriate to do so
  - Follow the guidelines given in today's lecture
- Add one more clever feature to your **`UnitConverter`** class!
  - If your converter knows how to convert from unit A to unit B, and from unit B to unit C, then we should also support converting from unit A to unit C