CS11 Intro C++

Spring 2018 – Lecture 2

C++ Compilation

- You type:

```
g++ -Wall -Werror units.cpp convert.cpp -o convert
```

- What happens?

- C++ compilation is a multi-step process:
  - Preprocessing
  - Compilation
  - Linking

- Different steps have different kinds of errors
  - …very helpful to understand what is going on!

C++ Compilation: Overview

- For preprocessing and compilation phases, each source file is handled separately

```
g++ -Wall -Werror units.cpp convert.cpp -o convert
```

- Compiler performs preprocessing and compilation on **`units.cpp`** and **`convert.cpp`** separately
  - Produces **`units.o`** and **`convert.o`**

- The linking phase combines the results of the compilation phase
  - **`units.o`** and **`convert.o`** are combined into a single executable program named **`convert`**

The Preprocessor

- Step 1: The Preprocessor
  - Prepares source files for compilation
- Performs various text-processing operations on each source file:
  - Removes all comments from the source file
  - Handles **preprocessor directives**, such as **`#include`** and **`#define`**
- Example: **`units.cpp`**: **`#include "units.h"`**
- Preprocessor *removes* this line from **`units.cpp`**, and *replaces* it with the contents of the file **`units.h`**

- For each input source file (i.e. each **`.cpp`** file):
  - The preprocessor generates a **translation unit** - the input that the compiler actually compiles

The Compiler

- The compiler takes a translation unit, and translates it from C++ code into machine code
  - i.e. from instructions that human beings understand, into instructions that your processor understands

- Result is called an **object file**
  - (Technically, it's called a "relocatable object file," but everyone just calls it an "object file")
  - e.g. **`units.o`** and **`convert.o`**

- These are not runnable programs, but they contain the machine-code instructions from your program

The Compiler: Object Files

- Object files are incomplete! They specify, among other things:

- Each function that is defined within the translation unit, along with its machine code
  - e.g. **`units.o`** contains a definition of "**`UValue convert_to(...)`**"
  - This includes the function's actual machine-code instructions

- Each function that is referred to by the translation unit, but whose definition is not specified
  - e.g. **`convert.o`** uses **`convert_to()`**, but doesn't have a definition of the function

The Linker

- The linker takes the object files generated by the compiler, and combines them together
- Many object files refer to functions that they don't actually implement

- Linker makes sure that every required function is defined in *some* object file

- Two main kinds of errors:
  - Linker can't find any definition of a function
  - Linker finds multiple definitions of a function!

Linker Errors

- Example: you forget to include **`main()`**
- Example output on Mac OS X:

```
Undefined symbols for architecture x86_64:
  "_main", referenced from:
     implicit entry/start for main executable
ld: symbol(s) not found for architecture x86_64
```

- **`ld`** is the linker program used by **`g++`**

- These errors don't occur during compilation
  - Compilation has succeeded, but the linker can't find definitions for some functions

Final Compilation Notes

- Generally, compilers don't leave intermediate files around anymore
  - They use much more efficient ways of passing translation units and object files to each other

- Can compile a source file without linking it:

  `g++ -Wall -Werror -c units.cpp`
  - Performs preprocessing and compilation
  - Produces **`units.o`**

- Can save other output files from preprocessor and compiler

  `g++ -Wall --save-temps -c units.cpp`
  - **`units.ii`** is result of running the preprocessor
  - **`units.s`** is a text version of the processor instructions

Units-Converter, Round 1

- First lab focused on writing a simple units-converter
  - Used a **`UValue`** class to package a value and its units together
- A number of issues in the implementation, in the **`convert_to()`** function
- Issue 1: Function hard-codes the unit conversions we can perform

```
if (from_units == "mi" && to_units == "km")
    return UValue{from_value * 1.6, to_units};
else if (from_units == "lb" && to_units == "kg")
    return UValue{from_value * 0.45, to_units};
else if (from_units == "gal" && to_units == "l")
    return UValue{from_value * 3.79, to_units};
else
    return v;
```

- Issue 2: Don't have a good way to report a failed conversion

Units-Converter, Round 2

- Would like to make our units-converter much more flexible
  - Easier to add unit-conversions to the program
- If we could keep a table of unit-conversion details, could look up conversions to perform from our table
- Fortunately, C++ provides a large number of useful collections in the C++ Standard Library
- Example: **`std::vector`** is dynamically-resizeable, growable array
  - (Just like C++ **`std::string`** is a dynamically-resizeable, growable string)
- The concept of a "vector" is independent of the element type…
- C++ provides vectors as a **class-template**
  - A class-template is not a class…
  - It is a generic, parameterized pattern for creating classes
  - **`std::vector<T>`** class-template takes the element-type as a parameter

C++ **`std::vector<T>`** (1)

- Vectors have a specific number of elements, reported by **`size()`**

```
vector<int> v1;          // Has 0 elements initially
vector<string> v2(10);   // Has 10 elements initially
```

- Can access and mutate elements using array-index operator []
  - Valid elements have indexes in the range 0 .. size() - 1
  - Warning: If you access an invalid element, you won't be stopped!
- Can use **`push_back(T)`** member function to append new values
  - Cost of appending is constant-time (amortized)
  - Vector maintains extra space at the end, to facilitate this
- Many other operations provided by **`vector<T>`**!
  - If you need a growable array, use this type!

C++ **std::vector<T>** (2)

- Example usage:

```
#include <vector>

vector<int> v;  // A vector that holds int elements
// Put some values into the vector
v.push_back(15);
v.push_back(42);
v.push_back(-9);
cout << "Number of elements:  " << v.size() << "\n";
for (int i = 0; i < v.size(); i++)
    cout << "v[" << i << "] = " << v[i] << "\n";
```

- Outputs:

```
Number of elements:  3
v[0] = 15
v[1] = 42
v[2] = -9
```

C++ **std::vector<T>** (3)

- If you don't care about printing the indexes:

```
vector<int> v;  // A vector that holds int elements
// Put some values into the vector
v.push_back(15);
v.push_back(42);
v.push_back(-9);
cout << "Number of elements:  " << v.size() << "\n";
for (int n : v)
    cout << " " << n;
cout << "\n";
```

  - A simple example of C++11 range-based for loop
- Outputs:

```
Number of elements:  3
 15 42 -9
```

C++ and Structs

- C++ includes structs as well as classes

- Main difference:
  - Struct members are public by default; class members are private by default
  - Can use access-modifiers in structs, just like classes
  - Can write constructors, destructors, member functions, operator overloads on structs, just like classes

- Generally, structs are used when the full functionality of classes isn't required
  - e.g. just need a heterogeneous data type to hold some data values

C++ and Structs (2)

- Example: a todo-list class

```
class TodoList {
    ...
public:
    int add_task(string description);
    void complete_task(int task_id);
};
```

- The class must keep track of each task's ID, description, and whether the task has been completed

```
struct TodoItem {
    int id;
    string description;
    bool completed;
};
```

  - All struct members are public access

C++ and Structs (3)

- **`TodoList`** class can use **`TodoItem`** struct to record task details

```
struct TodoItem {
    int id;
    string description;
    bool completed;
};

class TodoList {
    int next_id;
    vector<TodoItem> items;
public:
    int add_task(string description);
    void complete_task(int task_id);
};
```

  - External interface remains clean and simple
  - Use of this struct is hidden from users of the class

C++ and Structs (4)

- C++ allows class/struct declarations to be nested

```
class TodoList {
    struct TodoItem {
        int id;
        string description;
        bool completed;
    };

    int next_id;
    vector<TodoItem> items;
public:
    int add_task(string description);
    void complete_task(int task_id);
};
```

  - **`TodoItem`** type is in private section of **`TodoList`**
  - Now, **`TodoItem`** type isn't even visible to code outside of **`TodoList`** class

C++ and Structs (5)

- Can specify initial values for structs, just as in C

```
int TodoList::add_task(string description) {
    TodoItem i = {next_id, description, false};
    items.push_back(i);
    ++next_id;
}
```

- Could even write:

```
items.push_back({next_id, description, false});
```

Reporting and Handling Failed Operations

- Consider this function:

```
double compute_value(double x) {
    return sqrt(x - 3.0);
}
```

- Will it work for all inputs?
  - Only works for inputs x >= 3.0

- How to indicate when the computation fails?
- Could use a special return-value… (gross)
  - Callers must know what value means “the computation failed”…
  - Callers must check the result for this special value

C++ Exceptions

- C++ includes support for **exception handling**
- Code that detects an error, but doesn't know what to do about it, can **throw** an exception
- Code that can handle the error, but can't detect it, can **catch** the exception
  - May be the immediate caller of the function, or may be separated by many function invocations
- The exception's type indicates the category of the error/failure
  - e.g. **`out_of_range`** for index-access functions that receive a bad index
  - e.g. **`regex_error`** for problems in evaluating regular expressions
- In C++, anything may be thrown as an exception…
  - (but please don't! ☺)
  - Usually, specific classes are created to indicate specific kinds of errors

Reporting a Failure

- Our function can indicate when there is a problem:

```
double compute_value(double x) {
    if (x < 3.0)
        throw invalid_argument("x must be >= 3");

    return sqrt(x - 3.0);
}
```

- Now our function can complete in two ways
- Normal completion: function computes and returns sqrt(x – 3)
- Abnormal termination: function detects an error and aborts the computation
- Many exception classes include state to report nature of the failure
  - Caller can use this state to determine the exact nature of the failure

Handling a Failure

- Callers can now be informed when the computation fails:

```
double x;
cout << "Enter x:  ";
cin >> x;
try {
    double v = compute_value(x);
    cout << "Answer is " << v << "\n";
}
catch (invalid_argument) {
    cout << "Error occurred!\n";
}
```

- If code in the try-block throws an **`invalid_argument`** exception, we will handle it!
  - If any other exception is thrown, we don't want to (or can't) handle it…

Handling a Failure (2)

- Callers can now be informed when the computation fails:

```
double x;
cout << "Enter x:  ";
cin >> x;
try {
    double v = compute_value(x);
    cout << "Answer is " << v << "\n";
}
catch (invalid_argument) {
    cout << "Error occurred!\n";
}
```

- If code in the try-block throws an **`invalid_argument`** exception, execution transfers immediately to the corresponding catch-block
  - The function doesn't complete. The "Answer is" output is also skipped.

Handling a Failure (3)

- Can give the caught exception a name, to access its details:

```
double x;
cout << "Enter x:  ";
cin >> x;
try {
    double v = compute_value(x);
    cout << "Answer is " << v << "\n";
}
catch (invalid_argument e) {
    cout << "Error occurred!\n";
    cout << e.what() << "\n";
}
```

- Most C++ standard exceptions include a “what” value for reporting what happened

Functions and Exceptions

- Exceptions are as much a part of a function's public interface, as the arguments and the return-value!
- *Extremely important* to document what exceptions are thrown, and the circumstances in which they are thrown.

- Example:

```
/* Given x, computes the square-root of x - 3.
 *
 * Throws invalid_argument if x < 3.
 */
double compute_value(double x) {
    ...
}
```

This Week's Homework

- Implement a much more data-driven unit-conversion program
- Make a **`UnitConverter`** class that manages a collection of unit-conversions, using a nested **`struct`** and a **`std::vector`** data-member
- Throw exceptions to report various failures
- Main program is updated to use the **`UnitConverter`**, and to handle exceptions that can be thrown

- A test program will also be supplied to exercise your **`UnitConverter`** code