CENG 112 – Data Structures

Introduction

Mustafa Özuysal
mustafaozuysal@iyte.edu.tr

February 24, 2017

İzmir Institute of Technology

Course Information

Course Contents

This course covers the fundamental data structures such as linked lists and trees and algorithms for searching and sorting. It also introduces the abstract data types and various ways to implement them. It includes the basics of algoritmic complexity analysis.

People and Office Hours

Assist. Prof. Mustafa Özuysal	D133-A	Thursday 13:30–14:30
Ersin Çine	D140	Thursday 9:00–12:00
Ali Köksal	D141	Thursday 13:30–17:00
Eren Uzyıldırım	D141	Friday 13:30–17:00

Grading

5% Attendance (Taken at random weeks)
25% Homeworks
35% Midterm Exam
35% Final Exam

There will be six homeworks in total, the best five grades will be taken into account. Homeworks will be distributed in two parts and collected every week.

Textbooks

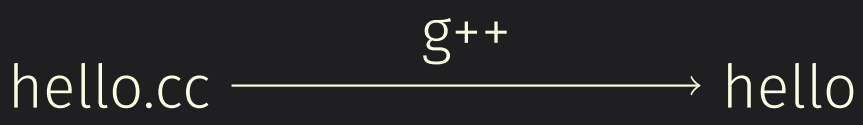
- Algorithms in C++ (3^{rd} Ed.), R. Sedgewick
- C++ Primer Plus (6^{th} Ed.), Stephen Prata

Introduction to C++

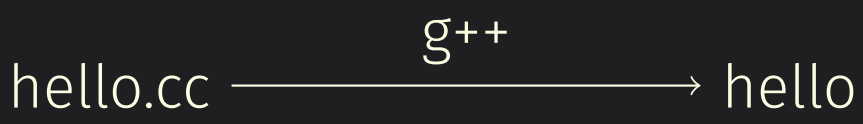
Hello World in C++

```cpp
// filename: hello.cc
// This is a C++ style single-line comment.
// The following includes a header file containing
// the definition of std::cout and std::endl.
#include <iostream>

// This is necessary so that we do not have to write std::cout.
using namespace std;

// This is the main function where execution will start.
int main(int argc, char* argv[]) {
        // This outputs Hello World! followed by a newline.
        cout << "Hello World!" << endl;

        // Returning zero means no error
        return 0;
}
```

Compiling and Executing



Compiling and Executing




```
# Lines starting with # are comments
# Lines starting with $ are what you type in a terminal
# -Wall, -g, -o are called command line arguments
# They let you customize what g++ compiler does
# -Wall enables all warnings
# -g adds debug symbols
# -o sets the executable name (default is a.out)
$ g++ -Wall -g hello.cc -o hello
# ./ at the front means in the current directory
$ ./hello
Hello World!
$
```

The `main`

- The `main` function is where the executables start running (actually this is only partially true).

The `main`

- The `main` function is where the executables start running (actually this is only partially true).
- It returns an integer, zero for success, non-zero for failure (usually an error code).

The `main`

- The `main` function is where the executables start running (actually this is only partially true).
- It returns an integer, zero for success, non-zero for failure (usually an error code).
- `argc` is the number of the command line arguments plus one.

The `main`

- The `main` function is where the executables start running (actually this is only partially true).
- It returns an integer, zero for success, non-zero for failure (usually an error code).
- `argc` is the number of the command line arguments plus one.
- `argv` is an array of C strings containing the command line arguments. `argv[0]` is the command that launched the program.

Printing and `for` Loops

```
1  // filename: args.cc
2  #include <cstdlib>
3  #include <iostream>
4
5  using namespace std;
6
7  int main(int argc, char* argv[])
8  {
9      // This for loops iterates as 0, 1, ..., (argc-1).
10     // Each iteration prints one command line argument on a
11     // seperate line.
12     for (int i = 0; i < argc; i++)
13             cout << "[" << i << "] " << argv[i] << endl;
14
15     // <cstdlib> header contains the constant EXIT_SUCCESS
16     // which is zero. It also contains the constant
17     // EXIT_FAILURE which is non-zero.
18     return EXIT_SUCCESS;
19 }
```

Printing and `for` Loops

```
$ g++ -Wall -g args.cc -o args
$ ./args
[0] ./args
$ ./args a b 12 "asb ads"
[0] ./args
[1] a
[2] b
[3] 12
[4] asb ads
```

Hello World with Multiple Sources

Headers and the Preprocessor

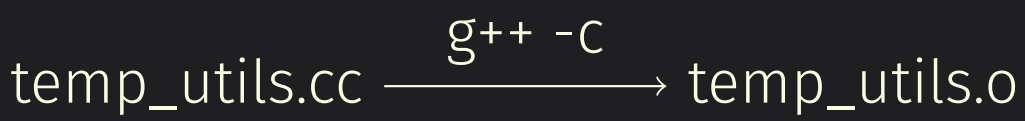
```
1  // filename: fahr2celc.cc
2  #include <cstdlib>
3  #include <cstring>
4  #include <iostream>
5  #include "temp_utils.h"
6  using namespace std;
7
8  int main(int argc, char** argv)
9  {
10     if (argc < 2) {
11             cerr << "Usage: " << argv[0]
12                  << " <temperature-in-F>" << endl;
13             return EXIT_FAILURE;
14     }
15
16     double temp_in_f = atof(argv[1]);
17     double temp_in_c = fahr_to_celcius(temp_in_f);
18     cout << temp_in_f << " degrees Fahrenheit is "
19          << temp_in_c << " degrees Celcius" << endl;
20
21     return EXIT_SUCCESS;
22 }
```

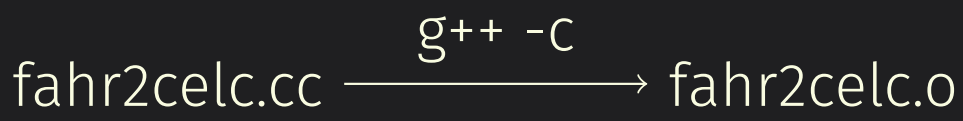
Headers and the Preprocessor

```
// filename: temp_utils.h
// This is called an include guard, it is necessary to avoid
// circular dependency chains between headers
#ifndef TEMP_UTILS_H
#define TEMP_UTILS_H

// Declaration of the function fahr_to_celcius
double fahr_to_celcius(double temp);

#endif
```

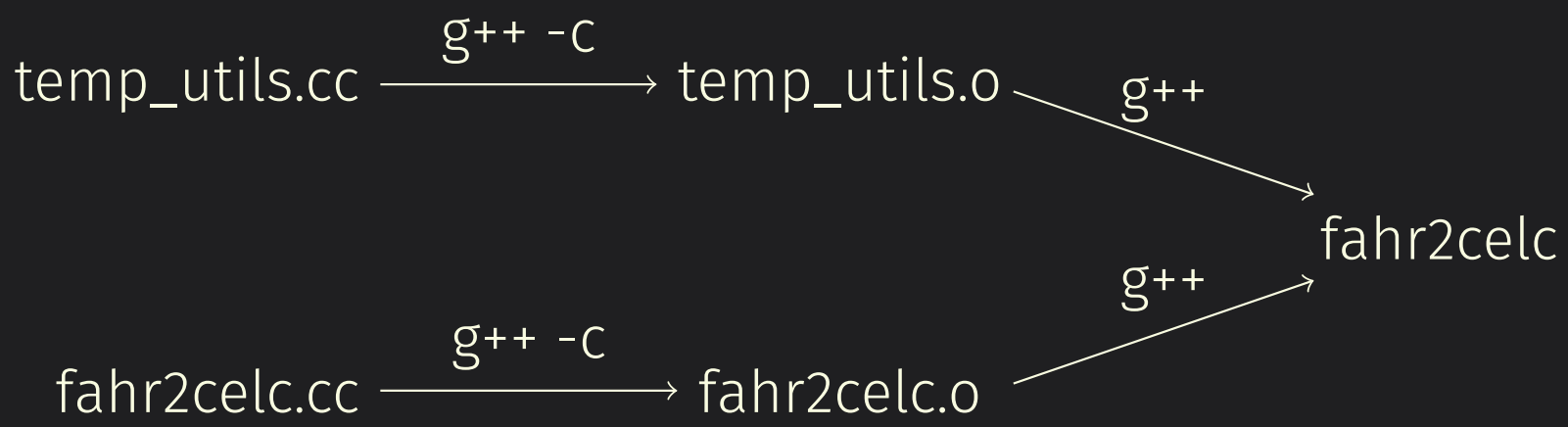
```
// filename: temp_utils.cc
#include "temp_utils.h"

// Definition of the function fahr_to_celcius
double fahr_to_celcius(double temp)
{
        return (temp - 32.0) * 5.0 / 9.0;
}
```

Compiling and Linking

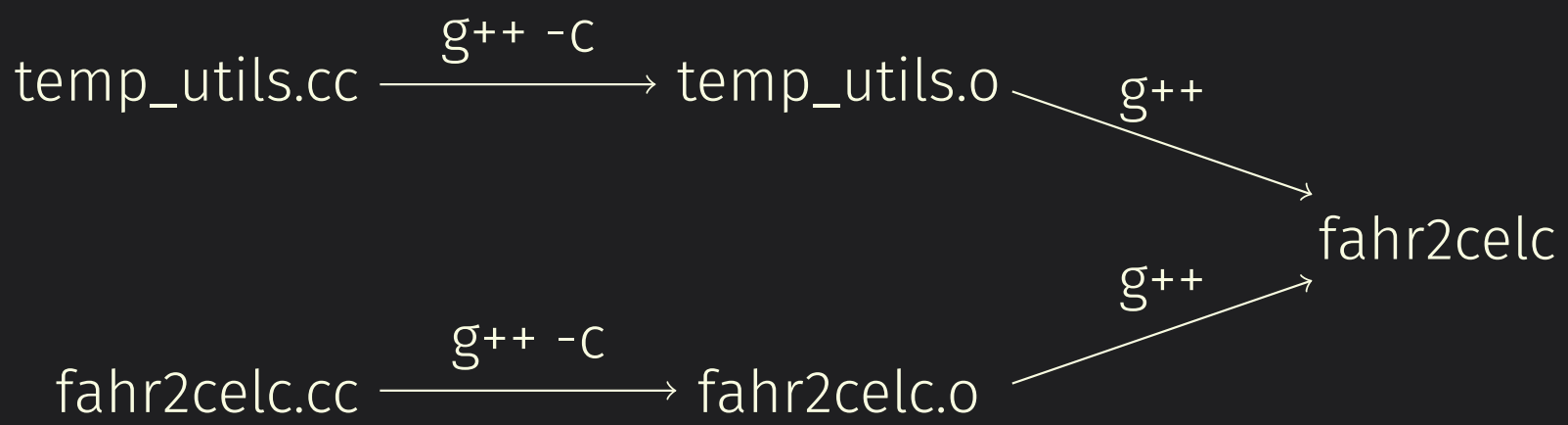




Compiling and Linking



Compiling and Linking




```
$ g++ -c -Wall -g fahr2celc.cc
$ g++ -c -Wall -g temp_utils.cc
$ g++ -Wall -g fahr2celc.o temp_utils.o -o fahr2celc
$ ./fahr2celc
Usage: ./fahr2celc <temperature-in-F>
$ ./fahr2celc 75
75 degrees Fahrenheit is 23.8889 degrees Celcius
$
```

Makefiles

Makefiles lets you specify the dependency chains for a list of targets and recipes to create the targets from their dependencies.

Makefiles

Makefiles lets you specify the dependency chains for a list of targets and recipes to create the targets from their dependencies.

```
<target>: <dependencies>
<tab-character><recipe>
```

Makefiles

Makefiles lets you specify the dependency chains for a list of targets and recipes to create the targets from their dependencies.

```
<target>: <dependencies>
<tab-character><recipe>
```

```
temp_utils.o: temp_utils.cc temp_utils.h
    g++ -c -g -Wall $< -o $@
```

In a recipe

- `$<` means the first dependency
- `$^` means all dependencies
- `$@` means target name

Makefiles

```
1  # filename: Makefile
2  CXXFLAGS:=-g -Wall
3
4  all: hello args fahr2celc
5
6  hello: hello.o
7          g++ ${CXXFLAGS} $^ -o $@
8
9  args: args.o
10         g++ ${CXXFLAGS} $^ -o $@
11
12 fahr2celc: fahr2celc.o temp_utils.o
13         g++ ${CXXFLAGS} $^ -o $@
14
15 temp_utils.o: temp_utils.cc temp_utils.h
16         g++ -c ${CXXFLAGS} $< -o $@
17
18 clean:
19         rm *.o *~ -f
```

A Build System: CMake

It is tedious to write Makefiles by hand. CMake can automatically generate a Makefile, a Visual Studio or Eclipse project from a simple definition file called CMakeLists.txt

A Build System: CMake

It is tedious to write Makefiles by hand. CMake can automatically generate a Makefile, a Visual Studio or Eclipse project from a simple definition file called CMakeLists.txt

```
1 # filename: CMakeLists.txt
2 project(ceng112_01 CXX)
3
4 add_executable(hello hello.cc)
5 add_executable(args args.cc)
6 add_executable(fahr2celc fahr2celc.cc temp_utils.cc)
7
8 set(CMAKE_BUILD_TYPE Debug)
```

A Build System: CMake

It is tedious to write Makefiles by hand. CMake can automatically generate a Makefile, a Visual Studio or Eclipse project from a simple definition file called CMakeLists.txt

```
# filename: CMakeLists.txt
project(ceng112_01 CXX)

add_executable(hello hello.cc)
add_executable(args args.cc)
add_executable(fahr2celc fahr2celc.cc temp_utils.cc)

set(CMAKE_BUILD_TYPE Debug)
```

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```