Programming in Modern C++

Tutorial T01: How to build a C/C++ program?: Part 1: C Preprocessor (CPP)

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Tutorial Objective

- How to build a C/C++ project?
- Understanding the differences and relationships between source and header files
- How C Preprocessor (CPP) can be used to manage code during build?

Tutorial Outline

Source and Header Files

Source Files

- **Source File**: A source file is a text file on disk. It contains instructions for the computer that are written in the C / C++ programming language
  - A source file typically has extension `.c` for C and `.cpp` for C++, though there are several other conventions
  - Any source file, called a *Translation Unit*, can be independently compiled into an object file (`*.o`)
  - A project may contain one or more source files
  - All object files of the project are linked together to create the executable binary file that we run
  - One of the source files must contain the `main()` function where the execution starts
  - Every source file includes zero or more header files to reduce code duplication
  - In a good source code organization, every header file has its source file that implements functions and classes. It is called *Implementation File*. In addition, *Application File*s would be there.

Header Files

- **Header File**: A header file is a text file on disk. It contains function declarations & macro definitions (C/C++) and class & template definitions (C++) to be shared between several source files
  - A header file typically has extension `.h` for C and `.h` or `.hpp` for C++, though there are several other conventions (or no extension for C++ Standard Library)
  - A header file is included in one or more source or header files
  - A header file is compiled as a part of the source file/s it is included in
    - Pre-Compiled Header (PCH): A header file may be compiled into an intermediate form that is faster to process for the compiler. Usage of PCH may significantly reduce compilation time, especially when applied to large header files, header files that include many other header files, or header files that are included in many translation units.
  - There are two types of header files. (More information in 19)
    - Files that the programmer writes are included as `#include "file"`
    - Files that comes with the compiler (*Standard Library*) are included as `#include <file>`. For C++
      - These have no extension and are specified within `std namespace`
      - The standard library files of C are prefixed with "`c`" with no extension in C++

Sample Source and Header Files in C

- **Header File**: `fact.h`: Includes the header for `fact()` function
- **Source File**: `fact.c`: Provides the implementation of `fact()` function
- **Source File**: `main.c`: Uses `fact()` function to compute factorial of given values

```
// File fact.h
// Header for Factorial function
#ifndef __FACT_H // Include Guard. Check
#define __FACT_H // Include Guard. Define

int fact(int);

#endif // __FACT_H // Include Guard. Close

// File fact.c
// Implementation of Factorial function
#include "fact.h" // User Header

int fact(int n) {
    if (0 == n) return 1;
    else return n * fact(n-1);
}
```

```
// File main.c
// Application using Factorial function
#include <stdio.h> // C Std. Library Header
#include "fact.h"  // User Header

int main() {
    int n, f;

    printf("Input n:"); // From stdio.h
    scanf("%d", &n);

    f = fact(n);

    printf("fact(%d) = %d", n, f); // From stdio.h

    return 0;
}
```

Sample Source and Header Files in C

- **Header File**: `Solver.h`: Includes the header for `quadraticEquationSolver()` function
- **Source File**: `Solver.c`: Provides the implementation of `quadraticEquationSolver()` function
- **Source File**: `main.c`: Uses `quadraticEquationSolver()` to solve a quadratic equation

```c
// File Solver.h
// User Header files
#ifndef __SOLVER_H // Include Guard. Check
#define __SOLVER_H // Include Guard. define
int quadraticEquationSolver(
    double, double, double, double*, double*);
#endif // __SOLVER_H // Include Guard. Close

// File Solver.c
// User Implementation files
#include <math.h>   // C Std. Library Header
#include "Solver.h" // User Header

int quadraticEquationSolver(
    double a, double b, double c, // I/P Coeff.
    double* r1, double* r2) {     // O/P Roots
    // Uses double sqrt(double) from math.h
    // ...
    return 0;
}
```

```c
// File main.c
// Application files
#include <stdio.h>  // C Std. Library Header
#include "Solver.h" // User Header

int main() {
    double a, b, c, r1, r2;
    // ...
    // Invoke the solver function from Solver.h
    int status = quadraticEquationSolver(
                 a, b, c, &r1, &r2);

    // int printf(char *format, ...) from stdio.h
    printf("Soln. for %dx^2+%dx+%d=0 is %d %d",
           a, b, c, r1, r2);
    // ...

    return 0;
}
```

Sample Source and Header Files in C++

- **Header File**: `Solver.h`: Includes the header for `quadraticEquationSolver()` function
- **Source File**: `Solver.cpp`: Provides the implementation of `quadraticEquationSolver()` function
- **Source File**: `main.cpp`: Uses `quadraticEquationSolver()` to solve a quadratic equation

```
// File Solver.h: User Header files
#ifndef __SOLVER_H // Include Guard. Check
#define __SOLVER_H // Include Guard. Define
int quadraticEquationSolver(
    double, double, double, double*, double*);
#endif // __SOLVER_H // Include Guard. Close

// File Solver.cpp: User Implementation files
#include <cmath>    // C Std. Lib. Header in C++
using namespace std;// C++ Std. Lib. in std
#include "Solver.h" // User Header

int quadraticEquationSolver(
    double a, double b, double c, // I/P Coeff.
    double* r1, double* r2) {     // O/P Roots
    // Uses double sqrt(double) from cmath
    // ...
    return 0;
}
```

```
// File main.c: Application file
#include <iostream>  // C++ Std. Library Header
using namespace std; // C++ Std. Lib. in std
#include "Solver.h"  // User Header

int main() {
    double a, b, c, r1, r2;
    // ...
    // Invoke the solver function from Solver.h
    int status = quadraticEquationSolver(
                 a, b, c, &r1, &r2);

    // From iostream
    cout<<"Soln. for "<<a<<"x^2+"<<b<<"x+"<<c"=0 is ";
    cout<< r1 << r2 << endl;
    // ...

    return 0;
}
```

C Preprocessor (CPP): Managing Source Code

Source: Preprocessor directives, cplusplus.com Accessed 13-Sep-21

C Preprocessor (CPP): Managing Source Code

- The CPP is the macro preprocessor for the C and C++. CPP provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control
- The CPP is driven by a set of directives
  - Preprocessor directives are lines included in the code of programs preceded by a #
  - These lines are not program statements but directives for the preprocessor
  - The CPP examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements
  - The CPP directives have the following characteristics:
    - CPP directives extend only across a single line of code
    - As soon as a newline character is found, the preprocessor directive is ends
    - No semicolon (;) is expected at the end of a preprocessor directive
    - The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\)

C Preprocessor (CPP): Macro definitions: #define, #undef

- To define preprocessor macros we can use `#define`. Its syntax is:

```
#define identifier replacement
```

- This replaces any occurrence of identifier in the rest of the code by replacement. CPP does not understand C/C++, it simply textually replaces

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
int table2[TABLE_SIZE];
```

- After CPP has replaced `TABLE_SIZE`, the code becomes equivalent to:

```
int table1[100];
int table2[100];
```

- We can define a symbol by `-D` *name* option from the command line. This predefines *name* as a macro, with definition 1. The following code compiles and outputs 1 when compiled with

```
$ g++ Macros.cpp -D FLAG

#include <iostream> // File Macros.cpp
int main() { std::cout << (FLAG==1) << std::endl; return 0; }
```

- **Note that `#define` is important to define constants (like size, pi, etc.), usually in a header (or beginning of a source) and use everywhere. `const` in a variable declaration is a better solution in C++ and C11 onward**

C Preprocessor (CPP): Macro definitions: #define, #undef

- `#define` can work also with parameters to define function macros:

```
#define getmax(a,b) a>b?a:b
```

- This replaces a occurrence of `getmax` followed by two arguments by the replacement expression, but also replacing each argument by its identifier, exactly as a function:

```
// function macro
#include <iostream>
using namespace std;

#define getmax(a,b) ((a)>(b)?(a):(b))

int main() {
    int x = 5, y;
    y= getmax(x,2);
    cout << y << endl << getmax(7,x) << endl;
    return 0;
}
```

- **Note that a `#define` function macro can make a small function efficient and usable with different types of parameters. In C++, inline functions & templates achieve this functionality in a better way**

C Preprocessor (CPP): Macro definitions: #define, #undef

- Defined macros are not affected by block structure. A macro lasts until it is undefined with the `#undef` preprocessor directive:

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
#undef TABLE_SIZE
#define TABLE_SIZE 200
int table2[TABLE_SIZE];
```

- This would generate the same code as:

```
int table1[100];
int table2[200];
```

- We can un-define a symbol by `-U` *name* option from the command line. This cancels any previous definition of *name*, either built in or provided with a `-D` option

```
$ g++ file.cpp -U FLAG
```

- **Note that `#undef` is primarily used to ensure that a symbol is not unknowingly being defined and used through some include path**

C Preprocessor (CPP): Macro definitions #define, #undef

- Parameterized macro definitions accept two special operators (`#` and `##`) in the replacement sequence: The operator `#`, followed by a parameter name, is replaced by a string literal that contains the argument passed (as if enclosed between double quotes):

```
#define str(x) #x
cout << str(test);
```

- This would be translated into:

```
cout << "test";
```

- The operator `##` concatenates two arguments leaving no blank spaces between them:

```
#define glue(a,b) a ## b
glue(c,out) << "test";
```

- This would also be translated into:

```
cout << "test";
```

- **Note that # and ## operators are primarily used in Standard Template Library (STL). They should be avoided at other places. As CPP replacements happen before any C++ syntax check, macro definitions can be a tricky. Code that relies heavily on complicated macros become less readable, since the syntax expected is on many occasions different from the normal expressions programmers expect in C++**

C Preprocessor (CPP): Conditional Inclusions: #ifdef, #ifndef, #if, #endif, #else & #elif

- These directives allow to include or discard part of the code of a program if a certain condition is met. This is known as **Conditional Inclusion** or **Conditional Compilation**
- `#ifdef` (*if defined*) allows a section of a program to be compiled only if the macro that is specified as the parameter has been `#define`, no matter which its value is. For example:

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

  In this case, the line of code `int table[TABLE_SIZE];` is only compiled if `TABLE_SIZE` was previously defined with `#define`, independently of its value. If it was not defined, that line will not be included in the program compilation
- `#ifndef` (*if not defined*) serves for the exact opposite: the code between `#ifndef` and `#endif` directives is only compiled if the specified identifier has not been previously defined. For example:

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

  In this case, if when arriving at this piece of code, the `TABLE_SIZE` macro has not been defined yet, it would be defined to a value of `100`. If it already existed it would keep its previous value since the `#define` directive would not be executed.

C Preprocessor (CPP): Conditional Inclusions: #ifdef, #ifndef, #if, #endif, #else & #elif

- The `#if`, `#else` and `#elif` (*else if*) directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows `#if` or `#elif` can only evaluate constant expressions, including macro expressions. For example:

```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```

- Notice how the entire structure of `#if`, `#elif` and `#else` chained directives ends with `#endif`
- The behavior of `#ifdef` and `#ifndef` can also be achieved by using the special operators `defined` and `!defined` (*not defined*) respectively in any `#if` or `#elif` directive:

```
#if defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
#elif !defined BUFFER_SIZE
#define TABLE_SIZE 128
#else
#define TABLE_SIZE BUFFER_SIZE
#endif
```

C Preprocessor (CPP): Typical Use-Cases
Conditional Inclusions: #ifdef, #ifndef, #if, #endif, #else & #elif

- **Commenting a large chunk of code**: We often need to comment a large piece of code. Doing that with C/C++-style comment is a challenge unless the Editor provides some handy support. So we can use:

```
#if 0 // "0" is taken as false and the codes till the #endif are excluded
Code lines to comment
#endif
```

- **Selective debugging of code**: We often need to put a lot of code the purpose of debugging which we do not want when the code is built for release with optimization. This can be managed by a `_DEBUG` flag

```
#ifdef _DEBUG
Code for debugging like print messages
#endif
```

Then we build the code for debugging as:

```
$ g++ -g -D _DEBUG file_1.cpp, file_2.cpp, ..., file_n.cpp
```

And we build the code for release as (`-U _DEBUG` may be skipped if there is no built-in definition):

```
$ g++ -U _DEBUG file_1.cpp, file_2.cpp, ..., file_n.cpp
```

- **Controlling code from build command line**: Suppose our project has support for 32-bit as well as 64-bit (*default*) and only one has to be chosen. So we can build for 32-bit using a flag `_BITS32`

```
$ g++ -D _BITS32 file_1.cpp, file_2.cpp, ..., file_n.cpp
```

And code as:

```
#ifndef _BITS32
Code for 64-bit
#else
Code for 32-bit
#endif
```

C Preprocessor (CPP): Source File Inclusion: #include

- When the preprocessor finds an `#include` directive it replaces it by the entire content of the specified header or file. There are two ways to use `#include`:

```
#include <header>
#include "file"
```

- In the first case, a header is specified between angle-brackets `<>`. This is used to include headers provided by the implementation, such as the headers that compose the standard library (`iostream, string, ...`). Whether the headers are actually files or exist in some other form is implementation-defined, but in any case they shall be properly included with this directive.
- The syntax used in the second `#include` uses quotes, and includes a file. The file is searched for in an implementation-defined manner, which generally includes the current path. In the case that the file is not found, the compiler interprets the directive as a header inclusion, just as if the quotes (`""`) were replaced by angle-brackets (`<>`)
- We can include a file by `-include` *`file`* option from the command line. So

```
using namespace std; // #include <iostream> skipped for illustration
int main() {
    cout << "Hello World" << endl;
    return 0;
}
```

would still compile fine with:

```
$ g++ "Hello World.cpp" -include iostream
```

C Preprocessor (CPP): Source File Inclusion: #include Guard

- Inclusions of header files may lead to the problems of Multiple Inclusion and / or Circular Inclusion
- An `#include` **guard**, sometimes called a **macro guard**, **header guard** or **file guard**, is a particular construct used to avoid the problem of double inclusion when dealing with the include directive
- **Multiple Inclusion**: Consider the following files:

Without Guard

```
// File "grandparent.h"
struct foo { int member; };

// File "parent.h"
#include "grandparent.h"

// File "child.c"
#include "grandparent.h"
#include "parent.h"

// Expanded "child.c": WRONG
// Duplicate definition
struct foo { int member; };
struct foo { int member; };
```

With Guard

```
// File "grandparent.h"
#ifndef GRANDPARENT_H // Undefined first time
#define GRANDPARENT_H // Defined for the first time
struct foo { int member; };
#endif /* GRANDPARENT_H */

// File "parent.h"
#ifndef PARENT_H
#define PARENT_H
#include "grandparent.h"
#endif /* PARENT_H */

// File "child.c"
#include "grandparent.h"
#include "parent.h"

// Expanded "child.c": RIGHT: Only one definition
struct foo { int member; };
```

C Preprocessor (CPP): Source File Inclusion: #include Guard

- **Circular Inclusion**: Consider the following files:

Without Guard

- Class Flight: Needs the info of service provider
- Class Service: Needs the info of flights it offers

```
#include<iostream>          // File main.h
#include<vector>
using namespace std;
#include "main.h"           // File Service.h
#include "Flight.h"
class Flight;
class Service { vector<Flight*> m_Flt; /* ... */ };
#include "main.h"           // File Flight.h
#include "Service.h"
class Service;
class Flight { Service* m_pServ; /* ... */ };
#include "main.h"           // File main.cpp
#include "Service.h"
#include "Flight.h"
int main() { /* ... */ return 0; };
```

- **Class Flight and Class Service has cross-references**
- **Hence, circular inclusion of header files lead to infinite loop during compilation**

With Guard

```
#include<iostream>          // File main.h
#include<vector>
using namespace std;
#ifndef __SERVICE_H
#define __SERVICE_H
#include "main.h"           // File Service.h
#include "Flight.h"
class Flight;
class Service { vector<Flight*> m_Flt; /* ... */ };
#endif // __SERVICE_H
#ifndef __FLIGHT_H
#define __FLIGHT_H
#include "main.h"           // File Flight.h
#include "Service.h"
class Service;
class Flight { Service* m_pServ; /* ... */ };
#endif // __FLIGHT_H
#include "main.h"           // File main.cpp
#include "Service.h"
#include "Flight.h"
int main() { /* ... */ return 0; };
```

C Preprocessor (CPP): Line control: #line and Error directive #error

- When we compile a program and some error happens during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.
- `#line` directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:

```
#line number "filename"
```

  Where number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.
  `"filename"` is an optional parameter that allows to redefine the file name that will be shown. For example:

```
#line 20 "assigning variable"
int a?;
```

  This code will generate an error that will be shown as error in file `"assigning variable", line 20`
- `#error` directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter:

```
#ifndef __cplusplus
#error A C++ compiler is required!
#endif
```

  This example aborts the compilation process if the macro name `__cplusplus` is not defined (this macro name is defined by default in all C++ compilers).

C Preprocessor (CPP): Pragma directive: #pragma

- This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with `#pragma`
- If the compiler does not support a specific argument for `#pragma`, it is ignored - no syntax error is generated
- Many compilers, including GCC, supports `#pragma once` which can be used as `#include` **guard**. So

```
#ifndef __FLIGHT_H
#define __FLIGHT_H
#include "main.h"       // File Flight.h
#include "Service.h"
class Service;
class Flight { Service* m_pServ; /* ... */ };
#endif // __FLIGHT_H
```

can also be written as:

```
#pragma once
#include "main.h"       // File Flight.h
#include "Service.h"
class Service;
class Flight { Service* m_pServ; /* ... */ };
```

This is cleaner, but may have portability issue across machines and compilers

C Preprocessor (CPP): Predefined Macro Names

- The following macro names are always defined (they begin and end with two underscore characters, _):

Macro	Value
__LINE__	Integer value representing the current line in the source code file being compiled
__FILE__	A string literal containing the presumed name of the source file being compiled
__DATE__	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began
__TIME__	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began
__cplusplus	An integer value. All C++ compilers have this constant defined to some value. Its value depends on the version of the standard supported by the compiler: • 199711L: ISO C++ 1998/2003 • 201103L: ISO C++ 2011 Non conforming compilers define this constant as some value at most five digits long. Note that many compilers are not fully conforming and thus will have this constant defined as neither of the values above
__STDC_HOSTED__	1 if the implementation is a hosted implementation (with all standard headers available) 0 otherwise

C Preprocessor (CPP): Predefined Macro Names

- The following macros are optionally defined, generally depending on whether a feature is available:

Macro	Value
__STDC__	In C: if defined to 1, the implementation conforms to the C standard. In C++: Implementation defined
__STDC_VERSION__	In C: • 199401L: ISO C 1990, Amendment 1 • 199901L: ISO C 1999 • 201112L: ISO C 2011 In C++: Implementation defined
__STDC_MB_MIGHT_NEQ_WC__	1 if multibyte encoding might give a character a different value in character literals
__STDC_ISO_10646__	A value in the form yyyymmL, specifying the date of the Unicode standard followed by the encoding of wchar_t characters
__STDCPP_STRICT_POINTER_SAFETY__	1 if the implementation has strict pointer safety (see get_pointer_safety)
__STDCPP_THREADS__	1 if the program can have more than one thread

- Macros marked in blue are frequently used

C Preprocessor (CPP): Standard Macro Examples

- Consider:

```
// standard macro names
#include <iostream>
using namespace std;

int main()
{
    cout << "This is the line number " << __LINE__;
    cout << " of file " << __FILE__ << ".\n";
    cout << "Its compilation began " << __DATE__;
    cout << " at " << __TIME__ << ".\n";
    cout << "The compiler gives a __cplusplus value of " << __cplusplus;
    return 0;
}
```

- The output is:

```
This is the line number 7 of file Macros.c.
Its compilation began Sep 13 2021 at 11:30:07.
The compiler gives a __cplusplus value of 201402
```

- **Note that `__LINE__`, `__FILE__`, `__DATE__`, and `__TIME__` important for details in error reporting**

Tutorial Summary

- Understood the differences and relationships between source and header files
- Understood how CPP can be harnessed to manage code during build