Programming in Modern C++

Module M01: Course Overview

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

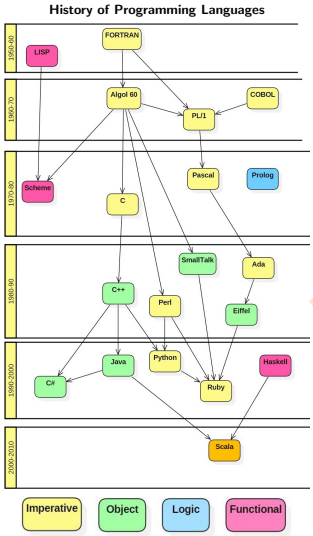
Module Objectives

- To understand the importance and ease of C++ in programming
- To Know Your Course including objective, prerequisites, outline, evaluation, books, and tools

Module Outline

Know Your C/C++

History of Programming Languages



Paradigms: *Imperative*: Algorithms + Data, *Object*: Data, *Logic*: Facts + Rules + Queries, and *Functional*: Functions

- **FORTRAN**: IBM
- **LISP**: John McCarthy
- **Algol 60**: John Backus & Peter Naur
- **COBOL**: Grace Murray Hopper
- **PASCAL**: Niklaus Emil Wirth
- **Prolog**: Alain Colmerauer & Philippe Roussel
- **Scheme**: Guy L. Steele & Gerald Jay Sussman
- **C**: Brian W. Kernighan & Dennis M. Ritchie
- **SmallTalk**: Alan Kay, Dan Ingalls, & Adele Goldberg
- **Ada**: Jean Ichbiah & Tucker Taft
- **C++**: Bjarne Stroustrup
- **Objective-C**: Brad Cox
- **Perl**: Larry Wall
- **Java**: James Gosling
- **Python**: Guido van Rossum
- **Haskell**: Paul Hudak
- **C#**: Microsoft Corporation
- **Ruby**: Yukihiro Matsumoto
- **Scala**: Martin Odersky

Source: Programming Language Evolution

TIOBE Index of Programming Languages: January 2021

Jan 2021	Jan 2020	Change	Programming Language	Ratings	Change
1	2	↑	C	17.38%	+1.61%
2	1	↓	Java	11.96%	-4.93%
3	3		Python	11.72%	+2.01%
4	4		C++	7.56%	+1.99%
5	5		C#	3.95%	-1.40%
6	6		Visual Basic	3.84%	-1.44%
7	7		JavaScript	2.20%	-0.25%
8	8		PHP	1.99%	-0.41%
9	18	⇈	R	1.90%	+1.10%
10	23	⇈	Groovy	1.84%	+1.23%
11	15	⇈	Assembly language	1.64%	+0.76%
12	10	↓	SQL	1.61%	+0.10%
13	9	⇊	Swift	1.43%	-0.36%
14	14		Go	1.41%	+0.51%
15	11	⇊	Ruby	1.30%	+0.24%
16	**20**	⇈	**MATLAB**	**1.15%**	**+0.41%**
17	19	↑	Perl	1.02%	+0.27%
18	13	⇊	Objective-C	1.00%	+0.07%
19	12	⇊	Delphi/Object Pascal	0.79%	-0.20%
20	16	⇊	Classic Visual Basic	0.79%	-0.04%

Top Programming Languages - IEEE Spectrum, index — TIOBE - The Software Quality Company

Choosing the Right Language

- Most systems need several languages for different parts of the system
  - HTML for front-end rendering and Javascript for active front-end logic
  - Java for servlet (business layer) and JSP for server-end embedding
  - SQL for data manipulation
- Nature of Application decides the choice of the language
  - Systems Programming $\Rightarrow$ C++ (very high performance with complex behavior)
  - Embedded Programming $\Rightarrow$ C (very high performance with frugal dev tools)
  - Application Programming $\Rightarrow$ Java (medium performance with quick & robust app)
  - Web Programming $\Rightarrow$ Python (low performance with portability)

Source: Why Undergraduates Should Learn the Principles of Programming Languages?, ACM SIGPLAN, 2011

Why learn C/C++?

- C++ is used in development of **Core Software**
  - *Databases*: Oracle, MySQL, MongoDB, MemSQL, etc. used for YouTube, Twitter, Facebook, etc.
  - *OS*: Windows, Linux, Android, Ubuntu, iOS, etc. are written in a combination of C and C++
  - *Compilers / VMs / Tools*: GNU Compiler Collection (GCC); JVM, PVM; MATLAB, IDE
  - *Web Browsers*: Chrome, Firefox, Safari, etc.
  - *Graphic Engine*: Applications in image processing, computer vision, screen recorders, games etc.
  - *Embedded Systems*: Smart watches, MP3 players, GPS systems, etc.
- C++ has **Core Strengths** like
  - *Fast*, *Portable*, and *Scalable*
  - Offers multiple levels of *Abstraction*: hardware to objects to meta-programs
  - *Multi-Paradigms*: Imperative / Procedural (C / Python), Object-Oriented (Algol / Java), Functional (LISP), Generic / Meta-Programming (template, lambda), Concurrent (Java)
- C++ has a **Large Community**
- C++ has **Abundant Library Support (STL)**
- C++ skills attract **High Salary**
- **Caveat**
  - It takes more time to be skilled in C++ compared to, say, Python due to its complexity and diversity
  - It is better to use Java / Python for simple front-end applications that are not performance critical
  - C++ is not best suited for front-end graphics applications for the lack of graphics library

Source: Top 10 Reasons to Learn C++, GeeksforGeeks, 2019

C Standards

K&R C	C89/C90	C95	C99	C11	C18
1978	1989/90	1995	1999	2011	2011
Created by Dennis Ritchie in early 1970s augmenting Ken Thompson's B	ANSI Std. in 1989	ISO Published Amendment	New built-in data types: `long long`, `_Bool`, `_Complex`, and `_Imaginary`	type generic macros	ISO Published Amendment
Brian Kernighan wrote the first C tutorial	ISO Std. in 1990	Errors corrected	Headers: `<stdint.h>`, `<tgmath.h>`, `<fenv.h>`, `<complex.h>`	Anonymous structures	Errors corrected
K & R published The C Programming Language in 1978. It worked as a defacto standard for a decade		Better multi-byte & wide character support in the library, with `<wchar.h>`, `<wctype.h>` and multi-byte I/O	static array indices, designated initializers, compound literals, variable-length arrays, flexible array members, variadic macros, and restrict keyword	Improved Unicode support	
ANSI C was covered in second edition in 1988		digraphs added	Compatibility with C++ like inline functions, single-line comments, mixing declarations and code, universal character names in identifiers	Atomic operations	
		Alternative specs. of operators, like 'and' for '&&'	Removed C89 language features like implicit function declarations and	Multi-threading	
		Std. macro `__STDC_VERSION__` with value `199409L` for C99 support		Std. macro `__STDC_VERSION__` defined as `201112L` for C11 support	Std. macro `__STDC_VERSION__` defined as `201710L` for C18 support
				Bounds-checked functions	
The C Programming Language, 1978	ANSI X3.159-1989 ISO/IEC 9899:1990	ISO/IEC 9899/ AMD1:1995	ISO/IEC 9899:1999	ISO/IEC 9899:2011	ISO/IEC 9899:2018

Latest Version as of Sep-21: C18: ISO/IEC 9899:2018, 2018

C++ Standards

C++98	C++11	C++14	C++17	C++20
1998	2011	2014	2017	2020
Templates	Move Semantics	Reader-Writer Locks	Fold Expressions	Coroutines
STL with Containers and Algorithms	Unified Initialization	Generic Lambda Functions	constexpr if	Modules
Strings	auto and decltype		Structured Binding	Concepts
I/O Streams	Lambda Functions		`std::string_view`	Ranges Library
	constexpr		Parallel Algortihms of the STL	
	Multi-threading and Memory Model		File System Library	
	Regular Expressions		`std::any`, `std::optional`, and `std::variant`	
	Smart Pointers			
	Hash Tables			
	`std::array`			
ISO/IEC 14882:1998	ISO/IEC 14882:2011	ISO/IEC 14882:2014	ISO/IEC 14882:2017	ISO/IEC 14882:2020

Fixes on C++98: C++03: ISO/IEC 14882:2003, 2003
Latest Version as of Sep-21: C++20: ISO/IEC 14882:2020, 2020

Know Your Course

Course Objectives

- Learn to develop software using C++ (C++98/03)
  - Features of C++ over and above C
  - Object-Oriented Paradigm in C++
  - STL for extensive code reuse
- Learn to improve software development using modern C++ (C++11)
  - Features of C++11 over and above C++98/03
  - Concurrent Programming in C++
  - Better quality and efficiency by C++11
- Cultivate skills to design, code, debug, and test software written in C++
- Attain strong employability with hands-on skills of software development

Course Prerequisites

Data Structures

- Array
- List
- Binary Search Tree
  - Balanced Tree
- B-Tree
- Hash Table / Map

Algorithms & Programming in C

- Sorting
  - Merge Sort
  - Quick Sort
- Search
  - Linear Search
  - Binary Search
  - Interpolation Search

Object-Oriented Analysis and Design

NPTEL Courses

- Design and Analysis of Algorithms
- Introduction to Programming in C
- Object-Oriented Analysis and Design

Quick Recap Modules

- Two self-study modules (QR1 & QR2) are provided for quick recap in Week 0
- Recap would be necessary before moving on to Module 02

Course Outline

- The course comprises:
  - 60 **Modules** (5 modules / week for 12 weeks). These are numbered serially as `Mnn`
    - These cover the course syllabus
    - These are used in assignments and examinations
  - Supplementary **Quick Recap** modules to revise C language and related topics in Week 0. These are numbered serially as `QRn`
    - These may be used to recapitulate C programming, as needed
    - These are not directly part of the syllabus, but cover the prerequisites. So their understanding are critical for the main modules. Those who know, may skip
  - **Tutorials** to build skills in C / C++ programming. These are numbered serially as `Tnn`
    - Some tutorials are of *Complementary* nature. These talk about various aspects of program development, program building, programming practices, etc. that may help to develop software using C / C++
    - Remaining tutorials are of *Supplementary* nature. These talk about additional information about C / C++ like how to mix these language, what is their compatibility etc.
    - Tutorials are not part of the syllabus. These are included for developing allround skills for those who desire so

Course Outline: Modules

Week	Topic
Week 01	**Programming in C++ is Fun**: *Introduction & Overview*
Week 02	**C++ is Better C**: *Procedural Extensions of C*
Week 03	**OOP in C++/1**: *Classes and Encapsulation*
Week 04	**OOP in C++/2**: *Overloading, namespace, struct & union*
Week 05	**Inheritance**: *ISA & HAS_A in C++*
Week 06	**Polymorphism**: *Binding, VFT, Multiple Inheritance*
Week 07	**Type Casting**: *C++ cast operators*
Week 08	**Exceptions & Templates**: try-throw-catch; *Meta-programming*
Week 09	**Streams & STL**: *IO, Containers, Algorithms*
Week 10	**Modern C++**: *C++11 and beyond – better C++, basic features*
Week 11	**λ & Concurrency**: *λ functions; threads, async call & mutex*
Week 12	**Move, Rvalue & Containers**: *Move semantics; Summarization*

C++98/03

C++11

Course Outline: Tutorials

- **Tutorials** are complementary or supplementary:
  - *Complementary* Tutorials introduce new ideas and skill areas to complement the understanding of the C/C++ languages. These include:
    - ▷ How to build a C/C++ program and / or static and dynamic libraries?
    - ▷ How to automate build using `make` utility?
    - ▷ What tools may be used to design, develop, test, and manage C / C++ software?
    - ▷ How to reuse?
      - — binary (static or dynamic library)
      - — code (template and meta-programming)
      - — design (desing pattern)
    - ▷ and more
  - *Supplementary* Tutorials provide additional information and insight to supplement the understanding of the C/C++ languages. These include:
    - ▷ How to mix C/C++ in a single program?
    - ▷ What is the compatibility of C/C++?
    - ▷ What are the coding styles to write good C/C++ programs?
    - ▷ and more

Course Evaluation

- **Assignments:** Once every week
  - Quiz Assignments
  - Programming Assignments
  - Weekly Assignment Score = Quiz Assignment Score + Programming Assignment score
  - Best 9 assignment scores (out of 12) to be considered for certification criteria
- **Unproctored Test:** 20 Marks
  - Type of questions: Programming. Very similar to the Programming assignments
  - You can appear the test from your home/college/work place itself using your PC (It may not support the mobile)
- **Proctored Test:** 80 Marks
  - Type of the questions: MCQ, MSQ, and short answer (SA) or one word type.
  - You need to visit the allocated exam center for this test
  - Online test (Computer based)
- **Certification Criteria**
  - All the scores are scaled to 100
  - Assignment score >= 40/100 AND Unproctored test score >= 40/100 AND Proctored test score >= 40/100

    (OR)

    Assignment score >= 10/25 AND Unproctored test score >= 10/25 AND Proctored test score >= 20/50
  - All the above three conditions have to be satisfied.
- *Note: NPTEL may change the certification criteria. However, You will get notified regarding the changes through an announcement prior to the tests. The evaluation process, marks distribution and certification criteria will be decided by the Instructor who runs the course in a specific semester.*

Textbooks, Tutorials, Standards, and Blogs

- **Textbooks**
  - The C Programming Language, Brian Kernighan and Dennis Ritchie, 1988 [Used here]
  - C programming: A Modern Approach, 2^{nd} Ed., Kim N. King, 2008
  - C++ Primer, 5^{th} Ed., S. Lippman, J. Lajoie, and B. Moo, 2012 [Most popular textbook]
  - Programming: Principles and Practice using C++, 2^{nd} Ed., Bjarne Stroustrup, 2014 [Used here]
  - The C++ Programming Language, 4^{th} Ed., Bjarne Stroustrup, 2013 [Authentic C++ Book]
- **Tutorials** [Free]
  - C Tutorial
  - Learn C and C++ Programming: C Tutorial [C], C++ Tutorial [C++]
  - LEARN C++: Skill up with our free tutorials [C++11, Used here]
- **Standards**
  - ISO C Standard: ISO/IEC 9899:2018 [Latest Standard]
  - ISO C++ Standards: ISO/IEC 14882:2020 [Latest Standard]
  - C++98 and C++03, C++11, C++14, C++17, C++20 [Free: Used here]
- **Blogs** [Free & Used here]
  - Bjarne Stroustrup: Creator of C++
  - Andrei Alexandrescu: Creator of D
  - Scott Meyers: Prolific educator of C++
  - Herb Sutter: Sutter's Mill: Chair of ISO C++ standards committee for over a decade

References

- **C++98/03**
  - Effective C++, 3rd Ed., 2005 and More Effective C++, 1st Ed., 1996, Scott Meyers [Used here]
  - Modern C++ Design, Andrei Alexandrescu, 2001 [Used here]
  - Exceptional C++, 1999 and More Exceptional C++, 2001 by Herb Sutter
  - Effective STL, 1st Ed., Scott Meyers, 2001
  - C++ Coding Standards, 1st Ed., Herb Sutter and Andrei Alexandrescu, 2004 [Used here]
  - The D Programming Language, Andrei Alexandrescu, 2010 [Future of C Family?]
  - Google C++ Style Guide
- **C++11, ...**
  - Effective Modern C++, Scott Meyers, 2015 [Used here]
  - Overview of the New C++ (C++11/14), Scott Meyers, 2015 [Used here]
  - C++ Move Semantics - The Complete Guide, Nicolai M. Josuttis, 2020
  - C++ Concurrency in Action, 2nd Ed., Anthony Williams, 2019
  - C++17 - The Complete Guide, Nicolai M. Josuttis, 2020
  - C++17 In Detail, Bartlomiej Filipek, 2019
  - Professional C++, 4th Ed., Marc Gregoire, 2018
  - Functional Programming in C++, Ivan Čukić, 2018
  - C++ Templates, 2nd Ed., D. Vandevoorde, N. M. Josuttis, and D. Gregor, 2017
  - The C++ Standard Library: A Tutorial and Reference, 2nd Ed., Nicolai M. Josuttis, 2012

Tools

- MinGW - Minimalist GNU for Windows [Free & Downloadable. Used here]
  - A native Windows port of the GNU Compiler Collection (GCC), with freely distributable import libraries and header files for building native Windows applications
  - Use GDB: The GNU Project Debugger for code debugging
  - Check How to install gdb in windows 10 to install minGW and gdb for Windows together
- GNU Online Compiler [Free & Online]
  - From Language Drop-down, choose C (C99), C++ (C++11), C++14, C++17
  - To mark the language for gcc compilation, set `-std=<compiler_tag>`
    - Tags for C are: `c89`, `c90`, `c11`, `c17`, `c18`, etc. Further `-ansi` means `-std=c90`
    - Tags for C++ are: `c++98`, `c++03`, `c++11`, `c++14`, `c++17`, `c++20`, etc. Further `-ansi` means `-std=c++98`
    - Check 3.4 Options Controlling C Dialect and 2 Language Standards Supported by GCC for details and options
- Code::Blocks [Free & Online]
  - A free, open source cross-platform IDE that supports GCC, Clang, Visual C++, and others
  - Choose language flag based on the choice of compiler (check on the manual)
- Programiz Online Compiler [Free & Online]
  - Supports C18 and C++14
- OneCompiler [Free & Online]
  - Supports C11 and C++14
- *While using a compiler, make sure that you know the language version you are compiling for*

Tools: Checking Compiler Version

- Check __cplusplus macro in C++:

```
#include <iostream>
#include <typeinfo>
int main() {
    if (__cplusplus == 201703L) std::cout << "C++17\n";
    else if (__cplusplus == 201402L) std::cout << "C++14\n";
    else if (__cplusplus == 201103L) std::cout << "C++11\n";
    else if (__cplusplus == 199711L) std::cout << "C++98\n";
    else std::cout << "pre-standard C++\n";
}
```

- Check __STDC_VERSION__ macro in C:

```
#include <stdio.h>
int main() {
    if (__STDC_VERSION__ == 201710L) printf("C18\n");       // C11 with bug fixes
    else if (__STDC_VERSION__ == 201112L) printf("C11\n");
    else if (__STDC_VERSION__ == 199901L) printf("C99\n");
    else if (__STDC_VERSION__ == 199409L) printf("C89\n");
    else printf("pre-standard C\n");
}
```

Source: 3.7.1 Standard Predefined Macros

Module Summary

- Understood the importance and ease of C++ in programming
- Learnt about the course - objective, prerequisites, outline, evaluation, books, and tools

Programming in Modern C++

Module M02: IO & Loop

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

Module Recap

- Understood the importance and ease of C++ in programming
- Learnt about the course - objective, prerequisites, outline, evaluation, books, and tools

Module Objectives

- Understand differences between C and C++ programs
- Appreciate the ease of programming in C++

Note that here we are trying to understand the difference between the C-style of programming with the C++-style of programming, and how the C++ features make programming easier and less error-prone compared to its C equivalent. This is different from the compatibility issues between the two languages which will be discussed in Tutorial on **Compatibility of C and C++** *along with cross-functionality issues.*

Module Outline

1. Hello World: Handling IO
2. Add Two Numbers and Handling IO
3. Square Root: `math` Library
4. C and C++ Standard Library Headers & `std`
   - Header Conventions
5. Sum of `n` Numbers: Variable Declaration
6. Using Boolean in C and C++
7. Module Summary

Hello World: Handling IO

Program 02.01: Hello World

C Program

```c
// HelloWorld.c
#include <stdio.h>

int main() {
    printf("Hello World in C");
    printf("\n");

    return 0;
}
```

Hello World in C

- IO Header is `stdio.h`
- `printf` to *print* to console
- Console is `stdout` file
- `printf` is a variadic function
- `\n` to go to the new line
- `\n` is escaped newline character

C++ Program

```cpp
// HelloWorld.cpp
#include <iostream>

int main() {
    std::cout << "Hello World in C++";
    std::cout << std::endl;

    return 0;
}
```

Hello World in C++

- IO Header is `iostream`
- `operator<<` to *stream* to console
- Console is `std::cout` `ostream` (in `std` namespace)
- `operator<<` is a binary operator
- `std::endl` (in `std` namespace) to go to the new line
- `std::endl` is stream manipulator (newline) functor

Add Two Numbers and Handling IO

Program 02.02: Add two numbers

C Program

```c
// Add_Num.c
#include <stdio.h>
int main() { int a, b; int sum;

    printf("Input two numbers:\n");
    scanf("%d%d", &a, &b);

    sum = a + b;

    printf("Sum of %d and %d", a, b);
    printf(" is: %d\n", sum);
}
```

```
Input two numbers:
3 4
Sum of 3 and 4 is: 7
```

- `scanf` to *scan* (*read*) from console
- Console is `stdin` file
- `scanf` is a variadic function
- Addresses of `a` and `b` needed in `scanf`
- All variables `a`, `b` & `sum` declared first (`K&R`)
- Formatting (`%d`) needed for variables

C++ Program

```c
// Add_Num_c++.cpp
#include <iostream>
int main() { int a, b;

    std::cout << "Input two numbers:\n";
    std::cin >> a >> b;

    int sum = a + b; // Declaration of sum

    std::cout << "Sum of " << a << " and " << b <<
        " is: " << sum << std::endl;
}
```

```
Input two numbers:
3 4
Sum of 3 and 4 is: 7
```

- `operator>>` to *stream* from console
- Console is `std::cin` `istream` (in `std` namespace)
- `operator>>` is a binary operator
- `a` and `b` can be directly used in `operator>>` operator
- `sum` may be declared when needed. Allowed from `C89` too
- Formatting is derived from type (`int`) of variables

Square Root: math Library

Program 02.03: Square Root of a number

C Program

```
// Sqrt.c
#include <stdio.h>
#include <math.h>


int main() { double x, sqrt_x;
    printf("Input number:\n");
    scanf("%lf", &x);

    sqrt_x = sqrt(x);

    printf("Sq. Root of %lf is:", x);
    printf(" %lf\n", sqrt_x);
}
```

```
Input number:
2
Square Root of 2.000000 is: 1.414214
```

- Math Header is `math.h` (C Standard Library)
- Formatting (`%lf`) needed for variables
- `sqrt` function from C Standard Library
- Default precision in print is 6

C++ Program

```
// Sqrt_c++.cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() { double x;
    cout << "Input number:" << endl;
    cin >> x;

    double sqrt_x = sqrt(x);

    cout << "Sq. Root of " << x;
    cout << " is: " << sqrt_x << endl;
}
```

```
Input number:
2
Square Root of 2 is: 1.41421
```

- Math Header is `cmath` (C Standard Library in C++)
- Formatting is derived from type (`double`) of variables
- `sqrt` function from C Standard Library
- Default precision in print is 5 (*different*)

C and C++ Standard Library Headers & std

namespace std for C++ Standard Library

C Standard Library	C++ Standard Library
• All names are global • stdout, stdin, printf, scanf	• All names are within std namespace • std::cout, std::cin • Use using namespace std; to get rid of writing std:: for every standard library name

W/o using

```
#include <iostream>


int main() {

    std::cout << "Hello World in C++"
              << std::endl;

    return 0;
}
```

W/ using

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

int main() {

    cout << "Hello World in C++"
         << endl;

    return 0;
}
```

Standard Library: C/C++ Header Conventions

	C Header	C++ Header
C Program	Use `.h`. Example: `#include <stdio.h>` *Names in global namespace*	Not applicable
C++ Program	Prefix `c`, no `.h`. Example: `#include <cstdio>` *Names in `std` namespace*	No `.h`. Example: `#include <iostream>`

- A C std. library header is used in C++ with prefix '`c`' and without the `.h`. These are in `std` namespace:

```
#include <cmath>  // In C it is <math.h>
...
std::sqrt(5.0);   // Use with std::
```

It is possible that a C++ program include a C header as in C. Like:

```
#include <math.h> // Not in std namespace
...
sqrt(5.0);        // Use without std::
```

This, however, is not preferred

- **Using `.h` with C++ header files, like `iostream.h`, is disastrous. These are deprecated. It is dangerous, yet true, that some compilers do not error out on such use. Exercise caution.**

Sum of n Numbers: Variable Declaration

Program 02.04: Sum of n natural numbers

C Program

```
// Sum_n.c
#include <stdio.h>

int main() {
    int n;
    int i;
    int sum = 0;

    printf("Input limit:\n");
    scanf("%d", &n);

    for (i = 0; i <= n; ++i)
        sum = sum + i;

    printf("Sum of %d", n);
    printf(" numbers is: %d\n", sum);
}
```

```
Input limit:
10
Sum of 10 numbers is: 55
```

- `i` must be declared at the beginning (`C89`)

C++ Program

```
// Sum_n_c++.cpp
#include <iostream>
using namespace std;
int main() {
    int n;

    int sum = 0;

    cout << "Input limit:" << endl;
    cin >> n;

    for (int i = 0; i <= n; ++i) // Local Decl.
        sum = sum + i;

    cout << "Sum of " << n ;
    cout << " numbers is: " << sum << endl;
}
```

```
Input limit:
10
Sum of 10 numbers is: 55
```

- `i` declared locally in `for` loop. Allowed from `C99` too

Using Boolean in C and C++

Program 02.05: Using bool

C Program

```
// bool.c
#include <stdio.h>
#define TRUE 1
#define FALSE 0

int main() {
    int x = TRUE;

    printf
        ("bool is %d\n", x);
}
```

bool is 1

- Using int and #define for bool
- Only way to have bool in K&R

C Program

```
// bool.c
#include <stdio.h>
#include <stdbool.h>


int main() {
    bool x = true;

    printf
        ("bool is %d\n", x);
}
```

bool is 1

- stdbool.h included for bool
- _Bool type & macros in C89 expanding:
  bool to _Bool
  true to 1
  false to 0
  __bool_true_false_are_defined to 1

C++ Program

```
// bool_c++.cpp
#include <iostream>

using namespace std;

int main() {
    bool x = true;

    cout <<
        "bool is " << x;
}
```

bool is 1

- No additional headers required

  bool is a built-in type
  true is a literal
  false is a literal

Module Summary

- Understanding differences between C and C++ for:
  - IO
  - Variable declaration
  - Standard Library
- C++ gives us more flexibility in terms of basic declaration and input / output
- Many C constructs and functions are simplified in C++ which helps to increase the ease of programming

Programming in Modern C++

Module M03: Arrays and Strings

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

Module Recap

- Understanding differences between C and C++ for:
  - IO
  - Variable declaration
  - Standard Library
  - bool
- C++ gives us more flexibility in terms of basic declaration and input / output
- Many C constructs and functions are simplified in C++ which helps to increase the ease of programming

Module Objectives

- Understand array usage in C and C++
- Understand `vector` usage in C++
- Understand `string` functions in C and `string` type in C++

Module Outline

1. Arrays & vectors
   - Array Implementations for fixed size array
   - Array Implementations for arbitrary sized array
   - vectors in C++
2. C-Style Strings and string type in C++
   - Concatenation of strings
   - More string operations
   - string.h
   - string class
3. Module Summary

Arrays and vectors

Program 03.01: Fixed Size Array

C Program

```
// Array_Fixed_Size.c
#include <stdio.h>

int main() {
    short age[4];

    age[0] = 23;
    age[1] = 34;
    age[2] = 65;
    age[3] = 74;

    printf("%d ", age[0]);
    printf("%d ", age[1]);
    printf("%d ", age[2]);
    printf("%d ", age[3]);

    return 0;
}
```

```
23 34 65 74
```

C++ Program

```
// Array_Fixed_Size_c++.cpp
#include <iostream>

int main() {
    short age[4];

    age[0] = 23;
    age[1] = 34;
    age[2] = 65;
    age[3] = 74;

    std::cout << age[0] << " ";
    std::cout << age[1] << " ";
    std::cout << age[2] << " ";
    std::cout << age[3] << " ";

    return 0;
}
```

```
23 34 65 74
```


- No difference between arrays in C and C++

Program 03.02: Fixed size large array in C

Hard-coded

```
// Array_Large_Size.c
#include <stdio.h>
#include <stdlib.h>


int main() { int arr[100], sum = 0, i;
    printf("Enter no. of elements: ");
    int count;
    scanf("%d", &count);

    for(i = 0; i < count; i++) {
        arr[i] = i;
        sum + = arr[i];
    }
    printf("Array Sum: %d", sum);
}
```

```
Enter no. of elements: 10
Array Sum: 45
```

- Hard-coded size

Using manifest constant

```
// Array_Macro.c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

int main() { int arr[MAX], sum = 0, i;
    printf("Enter no. of elements: ");
    int count;
    scanf("%d", &count);

    for(i = 0; i < count; i++) {
        arr[i] = i;
        sum + = arr[i];
    }
    printf("Array Sum: %d", sum);
}
```

```
Enter no. of elements: 10
Array Sum: 45
```

- Size by manifest constant

Arbitrary Size Array

This can be implemented in C (C++) in the following ways:

- **Case 1:** Declaring a large array with size greater than the size given by users in all (most) of the cases
  - Hard-code the maximum size in code
  - Declare a manifest constant for the maximum size
- **Case 2:** Using `malloc` (`new[]`) to dynamically allocate space at run-time for the array

Program 03.03: Fixed large array / vector

C (array & constant)

```c
// Array_Macro_c.c
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

int main() { int arr[MAX];
    printf("Enter no. of elements: ");
    int count, sum = 0, i;
    scanf("%d", &count);
    for(i = 0; i < count; i++) {
        arr[i] = i; sum + = arr[i];
    }
    printf("Array Sum: %d", sum);
}
```

```
Enter no. of elements: 10
Array Sum: 45
```

- MAX is the declared size of array
- No header needed
- arr declared as int []

C++ (vector & constant)

```cpp
// Array_Macro_c++.cpp
#include <iostream>
#include <vector>
using namespace std;
#define MAX 100

int main() { vector<int> arr(MAX); // Define-time size
    cout <<"Enter the no. of elements: ";
    int count, sum = 0;
    cin >>count;
    for(int i = 0; i < count; i++) {
        arr[i] = i; sum + = arr[i];
    }
    cout << "Array Sum: " << sum << endl;
}
```

```
Enter no. of elements: 10
Array Sum: 45
```

- MAX is the declared size of vector
- Header vector included
- arr declared as vector<int>

Program 03.04: Dynamically managed array size

C Program

```
// Array_Malloc.c
#include <stdio.h>
#include <stdlib.h>


int main() { printf("Enter no. of elements ");
    int count, sum = 0, i;
    scanf("%d", &count);

    int *arr = (int*) malloc
        (sizeof(int)*count);
    for(i = 0; i < count; i++) {
        arr[i] = i; sum + = arr[i];
    }
    printf("Array Sum:%d ", sum);
}
```

```
Enter no. of elements: 10
Array Sum: 45
```

- malloc allocates space using sizeof

C++ Program

```
// Array_Resize_c++.cpp
#include <iostream>
#include <vector>
using namespace std;

int main() { cout << "Enter the no. of elements: ";
    int count, sum=0;
    cin >> count;

    vector<int> arr;    // Default size
    arr.resize(count);  // Set resize
    for(int i = 0; i < arr.size(); i++) {
        arr[i] = i; sum + = arr[i];
    }
    cout << "Array Sum: " << sum << endl;
}
```

```
Enter no. of elements: 10
Array Sum: 45
```

- resize fixes vector size at run-time

C-Style Strings and string type in C++

Strings in C and C++

String manipulations in C and C++:

- C-String and `string.h` library
  - C-String is an array of `char` terminated by `NULL`
  - C-String is supported by functions in `string.h` in C standard library
- `string` type in C++ standard library
  - `string` is a type
  - With operators (like `+` for concatenation) it behaves like a built-in type
  - In addition, for functions from C Standard Library `string.h` can be used in C++ as `cstring` in `std` namespace

Program 03.05: Concatenation of Strings

C Program

```
// Add_strings.c
#include <stdio.h>
#include <string.h>


int main() { char str1[] = {'H','E','L','L','O',' ','\0'};
    char str2[] = "WORLD";
    char str[20];
    strcpy(str, str1);
    strcat(str, str2);

    printf("%s\n", str);
}
```

```
HELLO WORLD
```

- Need header `string.h`
- *C-String is an array of characters*
- String concatenation done with `strcat` function
- Need a copy into `str`
- `str` must be large to fit the result

C++ Program

```
// Add_strings_c++.cpp
#include <iostream>
#include <string>
using namespace std;

int main(void) { string str1 = "HELLO ";
    string str2 = "WORLD";


    string str = str1 + str2;

    cout << str;
}
```

```
HELLO WORLD
```

- Need header `string`
- `string` is a data-type in C++ standard library
- Strings are concatenated like addition of `int`

More Operations on Strings

Further,

- `operator=` can be used on strings in place of `strcpy` function in C
- `operator<=`, `operator<`, `operator>=`, `operator>` operators can be used on strings in place of `strcmp` function in C

Strings in C and C++: C Standard Library Functions

Function	Description	Used Frequently?
Copying: `memcpy`	Copy block of memory (function)	Yes
`memmove`	Move block of memory (function)	Yes
`strcpy`	Copy string (function)	Yes
`strncpy`	Copy characters from string (function)	
Concatenation: `strcat`	Concatenate strings (function)	Yes
`strncat`	Append characters from string (function)	
Comparison: `memcmp`	Compare two blocks of memory (function)	
`strcmp`	Compare two strings (function)	Yes
`strcoll`	Compare two strings using locale (function)	
`strncmp`	Compare characters of two strings (function)	
`strxfrm`	Transform string using locale (function)	
Searching: `memchr`	Locate character in block of memory (function)	Yes
`strchr`	Locate first occurrence of character in string (function)	Yes
`strcspn`	Get span until character in string (function)	
`strpbrk`	Locate characters in string (function)	
`strrchr`	Locate last occurrence of character in string (function)	
`strspn`	Get span of character set in string (function)	
`strstr`	Locate substring (function)	Yes
`strtok`	Split string into tokens (function)	Yes
Other: `memset`	Fill block of memory (function)	
`strerror`	Get pointer to error message string (function)	
`strlen`	Get string length (function)	Yes
Macros: `NULL`	Null pointer (macro)	Yes
Types: `size_t`	Unsigned integral type (type)	Yes

Strings in C and C++: C++ Standard Library `string` Class

- Strings are objects that represent sequences of characters
- The standard string class provides support for such objects with an interface similar to that of a standard container of bytes, but adding features specifically designed to operate with strings of single-byte characters
- The string class is an instantiation of the `basic_string` class template that uses `char` (that is, bytes) as its character type, with its default `char_traits` and `allocator` types

Function	Description (Member Function)	C Parallel
Member functions		
`(constructor)`	Construct string object (public)	Initialize `string` object with a C string
`(destructor)`	String destructor (public)	
`operator=`	String assignment (public)	`strcpy()`. `operator=` does shallow copy
Iterators		Iteration done explicitly by loop index
`begin`	Return iterator to beginning (public)	
`end`	Return iterator to end (public)	
`rbegin`	Return reverse iterator to reverse beginning (public)	
`rend`	Return reverse iterator to reverse end (public)	
`cbegin`	Return const_iterator to beginning (public)	
`cend`	Return const_iterator to end (public)	
`crbegin`	Return const_reverse_iterator to reverse beginning (public)	
`crend`	Return const_reverse_iterator to reverse end (public)	

Strings in C and C++: C++ Standard Library `string` Class

Function	Description (Member Function)	C Parallel
Capacity		
`size`	Return length of string (public)	`strlen()`
`length`	Return length of string (public)	`strlen()`
`max_size`	Return maximum size of string (public)	Fixed at allocation
`resize`	Resize string (public)	
`capacity`	Return size of allocated storage (public)	Need to be remembered in the code
`reserve`	Request a change in capacity (public)	
`clear`	Clear string (public)	`strcpy()` an empty string
`empty`	Test if string is empty (public)	`strlen() == 0`
`shrink_to_fit`	Shrink to fit (public)	
String operations		
`c_str`	Get C string equivalent (public)	C string from a `string` object
`data`	Get string data (public)	
`get_allocator`	Get allocator (public)	
`copy`	Copy sequence of characters from string (public)	`strncpy()`
`find`	Find content in string (public)	`strchr()`, `strstr()`
`rfind`	Find last occurrence of content in string (public)	
`find_first_of`	Find character in string (public)	`strchr()`
`find_last_of`	Find character in string from the end (public)	`strrchr()`
`find_first_not_of`	Find absence of character in string (public)	
`find_last_not_of`	Find non-matching character in string from the end (public)	
`substr`	Generate substring (public)	`strncpy()`
`compare`	Compare strings (public)	`strcmp()`

Strings in C and C++: C++ Standard Library `string` Class

Function	Description (Member Function)	C Parallel
Element access		
`operator[]`	Get character of string (public)	`operator[]`
`at`	Get character in string (public)	`operator[]`
`back`	Access last character (public)	Character at `strlen()-1`
`front`	Access first character (public)	Character at 0^{th} location
Modifiers		
`operator+=`	Append to string (public)	`strcat()`
`append`	Append to string (public)	`strcat()`
`push_back`	Append character to string (public)	Set character to `strlen()` and `NULL` to next location
`assign`	Assign content to string (public)	
`insert`	Insert into string (public)	
`erase`	Erase characters from string (public)	
`replace`	Replace portion of string (public)	
`swap`	Swap string values (public)	Character by character swapping between two arrays
`pop_back`	Delete last character (public)	Set location `strlen()-1` to `NULL`
Member constants		
`npos`	Maximum value for size_t (public static)	
Non-member function overloads		
`operator+`	Concatenate strings (global)	`strcat()`
relational operators	Relational operators for string (global)	`strcmp()` followed by tests for `-1`, `0`, `+1`
`swap`	Exchanges the values of two strings (global)	
`operator>>`	Extract string from stream (global)	format `%s`
`operator<<`	Insert string into stream (global)	format `%s`
`getline`	Get line from stream into string (global)	`getline()` in `<stdlib.h>`

Module Summary

- Working with variable sized arrays is more flexible with `vector`s in C++
- String operations are easier with C++ standard library

Programming in Modern C++

Module M04: Sorting and Searching

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

Module Recap

- Working with variable sized arrays is more flexible with `vector`s in C++
- String operations are easier with C++ standard library

Module Objectives

- Implementation of Sorting and Searching in C and C++

Module Outline

Sorting in C and C++

Program 04.01: Bubble Sort

C Program

```
#include <stdio.h>

int main() { int data[] = {32, 71, 12, 45, 26};
    int i, step, n = 5, temp;
    for(step = 0; step < n - 1; ++step)
        for(i = 0; i < n-step-1; ++i) {
            if(data[i] > data[i+1]) {
                temp = data[i];
                data[i] = data[i+1];
                data[i+1] = temp;
            }
        }

    for(i = 0; i < n; ++i)
        printf("%d  ", data[i]);
}
```

```
12 26 32 45 71
```

C++ Program

```
#include <iostream>
using namespace std;
int main() { int data[] = {32, 71, 12, 45, 26};
    int n = 5, temp;
    for(int step = 0; step < n - 1; ++step)
        for(int i = 0;i < n-step-1; ++i) {
            if (data[i] > data[i+1]) {
                temp = data[i];
                data[i] = data[i+1];
                data[i+1] = temp;
            }
        }

    for(int i = 0; i < n; ++i)
        cout << data[i] << " ";
}
```

```
12 26 32 45 71
```


- Implementation is same in both C and C++ apart from differences in header files, I/O functions explained in Module 02

Program 04.02: Using sort from standard library

C Program (Desc order)

```c
#include <stdio.h>
#include <stdlib.h> // qsort function

// compare Function Pointer
int compare(
    const void *a, const void *b) { // Type unsafe
    return (*(int*)a < *(int*)b);   // Cast needed
}
int main () { int data[] = {32, 71, 12, 45, 26};
    // Start ptr., # elements, size, func. ptr.

    qsort(data, 5, sizeof(int), compare);

    for(int i = 0; i < 5; i++)
        printf ("%d ", data[i]);
}
```

```
71 45 32 26 12
```

- `sizeof(int)` and `compare` function passed to `qsort`
- `compare` function is type unsafe & needs complicated cast

C++ Program (Desc order)

```cpp
#include <iostream>
#include <algorithm> // sort function
using namespace std;
// compare Function Pointer
bool compare(
    int i, int j) { // Type safe
    return (i > j); // No cast needed
}
int main() { int data[] = {32, 71, 12, 45, 26};
    // Start ptr., end ptr., func. ptr.

    sort(data, data+5, compare);

    for (int i = 0; i < 5; i++)
        cout << data[i] << " ";
}
```

```
71 45 32 26 12
```

- Only `compare` passed to `sort`. No size is needed
- Only Size is inferred from the type `int` of `data`
- `compare` function is type safe & simple with no cast

Program 04.03: Using default sort of `algorithm`

C++ Program (Asc Order)

```
// sort.cpp
#include <iostream>
#include <algorithm> // sort function
using namespace std;

int main () {
    int data[] = {32, 71, 12, 45, 26};

    sort(data, data+5);

    for (int i = 0; i < 5; i++)
        cout << data[i] << " ";

    return 0;
}
```

```
12 26 32 45 71
```


- Sort using the default sort function of algorithm library which does the sorting in ascending order only
- No `compare` function is needed

Searching in C and C++

Program 04.04: Binary Search

C Program

```
#include <stdio.h>
#include <stdlib.h> // bsearch function

// compare Function Pointer
int compare(
    const void * a, const void * b) { // Type unsafe
    if (*(int*)a<*(int*)b) return -1; // Cast needed
    if (*(int*)a==*(int*)b) return 0; // Cast needed
    if (*(int*)a>*(int*)b) return 1;  // Cast needed
}
int main () { int data[] = {1,2,3,4,5}, k = 3;

    if (bsearch(&k, data, 5, sizeof(int), compare))
        printf("found!\n");
    else printf("not found\n");
}
```

```
found!
```

- `compare` function is type unsafe & needs complicated cast

C++ Program

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

int main() { int data[] = {1,2,3,4,5}, k = 3;

    if (binary_search(data, data+5, k))
        cout << "found!\n";
    else cout << "not found\n";
}
```

```
found!
```

- No `compare` function needed

STL: algorithm - The algorithm Library

The algorithm Library

The algorithm library of c++ helps us to easily implement commonly used complex functions. We discussed the functions for sort and search. Let us look at some more useful functions.

- Replace element in an array
- Rotates the order of the elements

Program 04.05: replace and rotate functions

Replace

```
// Replace.cpp
#include <iostream>
#include <algorithm> // replace function
using namespace std;

int main() {
    int data[] = {1, 2, 3, 4, 5};

    replace(data, data+5, 3, 2);

    for(int i = 0; i < 5; ++i)
        cout << data[i] << " ";

    return 0;
}
```

```
1 2 2 4 5
```

- 3^{rd} element replaced with 2

Rotate

```
// Rotate.cpp
#include <iostream>
#include <algorithm> // rotate function
using namespace std;

int main() {
    int data[] = {1, 2, 3, 4, 5};

    rotate(data, data+2, data+5);

    for(int i = 0; i < 5; ++i)
        cout << data[i] << " ";

    return 0;
}
```

```
3 4 5 1 2
```

- Array circular shifted around 3^{rd} element

Module Summary

- Flexibility of defining *customised* sort algorithms to be passed as parameter to sort and search functions defined in the `algorithm` library
- Predefined optimised versions of these sort and search functions can also be used
- There are a number of useful functions like `rotate`, `replace`, `merge`, `swap`, `remove` etc. in `algorithm` library

Programming in Modern C++

Module M05: Stack and Common Data Structures / Containers

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

Module Recap

- Flexibility of defining *customised* sort algorithms to be passed as parameter to sort and search functions defined in the `algorithm` library
- Predefined optimised versions of these sort and search functions can also be used
- There are a number of useful functions like `rotate`, `replace`, `merge`, `swap`, `remove` etc. in `algorithm` library

Module Objectives

- Understanding implementation and use of stack in C
- Understanding stack in C++ standard library and its use
- Understanding common containers in C++ standard library

Module Outline

1. Stack in C
   - Common Applications of Stack in C
   - Reverse a String
   - Evaluate a Postfix Expressions
2. Stack in C++
   - Reverse a String
   - Evaluate a Postfix Expressions
3. Data Structures / Containers in C++
   - Containers in C++
4. Module Summary

Stack in C

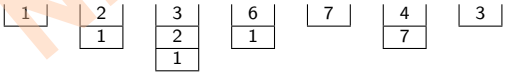
Stack in C

- Stack is a LIFO (last-In-First-Out) container that can maintain a collection of arbitrary number of data items – all of the same type
- To create a stack in C we need to:
  - Decide on the data type of the elements
  - Define a structure (container) (with maximum size) for stack and declare a `top` variable in the structure
  - Write separate functions for `push`, `pop`, `top`, and `isempty` using the declared structure
- **Note**:
  - Change of the data type of elements, implies re-implementation for all the stack codes
  - Change in the structure needs changes in all functions
- Unlike `sin`, `sqrt` etc. function from C standard library, we do not have a ready-made stack that we can use

Common C programs using stack

Some common C programs that use stack:

- *Reversing a string*
  - Input: ABCDE
  - Output: EDCBA
- *Evaluation of postfix expression*
  - Input: 1 2 3 * + 4 - (for 1 + 2 * 3 - 4)
  - Output: 3
    Stack states:



- *Identification of palindromes* (w/ and w/o center-marker)
- *Conversion of an infix expression to postfix*
- *Depth-first Search* (DFS)

Program 05.01: Reversing a string


```
#include <stdio.h>

typedef struct stack {
    char data [100];
    int top;
} stack;

int empty(stack *p) { return  (p->top  == -1); }

int top(stack *p) { return  p -> data [p->top]; }

void push(stack *p, char x) {
    p -> data [++(p -> top)] = x;
}

void pop(stack *p) {
    if (!empty(p)) (p->top) = (p->top) -1;
}
```

```
int main() {
    stack s;
    s.top = -1;

    char ch, str[10] = "ABCDE";

    int i, len = sizeof(str);

    for(i = 0; i < len; i++)
        push(&s, str[i]);

    printf("Reversed String: ");

    while (!empty(&s)) {
        printf("%c ", top(&s));
        pop(&s);
    }
}
```



```
Reversed String:  EDCBA
```

Program 05.02: Postfix Expression Evaluation

```
#include <stdio.h>

typedef struct stack {
    char data [100];
    int top;
} stack;

int empty(stack *p) {
    return (p->top  == -1);
}
int top(stack *p) {
    return p -> data [p->top];
}
void push(stack *p, char x) {
    p -> data [++(p -> top)] = x;
}
void pop(stack *p) {
    if (!empty(p)) (p->top) = (p->top) -1;
}
```

```
void main() { stack s; s.top = -1;

    // Postfix expression:  1 2 3 * + 4 -
    char postfix[] = {'1','2','3','*','+','4','-'};

    for(int i = 0; i < 7; i++) { char ch = postfix[i];
        if (isdigit(ch)) push(&s, ch-'0');
        else {
            int op2 = top(&s); pop(&s);
            int op1 = top(&s); pop(&s);
            switch (ch) {
                case '+': push(&s, op1 + op2); break;
                case '-': push(&s, op1 - op2); break;
                case '*': push(&s, op1 * op2); break;
                case '/': push(&s, op1 / op2); break;
            }
        }
    }
    printf("Evaluation %d\n", top(&s));
}
```


Evaluation 3

Stack in C++

Understanding Stack in C++

- C++ standard library provide a ready-made stack for any type of elements
- To create a stack in C++ we need to:
  - Include the `stack` header
  - Instantiate a stack with proper element type (like `char`)
  - Use the functions of the stack objects for stack operations

Program 05.03: Reverse a String in C++

```c
#include <stdio.h>
#include <string.h>
#include "stack.h" // User defined codes


int main() { char str[10] = "ABCDE";
    stack s; s.top = -1; // stack struct

    for(int i = 0; i < strlen(str); i++)
        push(&s, str[i]);

    printf("Reversed String: ");
    while (!empty(&s)) {
        printf("%c ", top(&s)); pop(&s);
    }
}
```

- *Lot of code* for creating stack in stack.h
- top to be initialized
- *Cluttered interface* for stack functions
- *Implemented by user* – *error-prone*

```c
#include <iostream>
#include <cstring>
#include <stack> // Library codes
using namespace std;

int main() { char str[10]= "ABCDE";
    stack<char> s; // stack class

    for(int i = 0; i < strlen(str); i++)
        s.push(str[i]);

    cout << "Reversed String: ";
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
}
```

- *No codes* for creating stack
- *No initialization*
- *Clean interface* for stack functions
- *Available in library* – *well-tested*

Program 05.04: Postfix Evaluation in C++

```
#include <iostream>
#include <stack> // Library codes
using namespace std;

int main() {
    // Postfix expression: 1 2 3 * + 4 -
    char postfix[] = {'1','2','3','*','+','4','-'}, ch;
    stack<int> s; // stack class

    for(int i = 0; i < 7; i++) { ch = postfix[i];
        if (isdigit(ch)) { s.push(ch-'0'); }
        else {
            int op1 = s.top(); s.pop();
            int op2 = s.top(); s.pop();
            switch (ch) {
                case '*': s.push(op2 * op1); break;
                case '/': s.push(op2 / op1); break;
                case '+': s.push(op2 + op1); break;
                case '-': s.push(op2 - op1); break;
            }
        }
    }
    cout << "\nEvaluation " << s.top();
}
```

Data Structures / Containers in C++

Data Structures / Containers in C++

- Like `Stack`, several other data structures are available in C++ standard library
- They are *ready-made* and *work like a data type*
- *Varied types of elements* can be used for C++ data structures
- **Data Structures** in C++ are commonly called **Containers**:
  - A container is a *holder object* that stores a *collection of other objects* (its elements)
  - They are implemented as *class templates* allowing great flexibility in the types supported as elements
  - The container
    - *manages the storage space* for its elements
    - provides member *functions to access* them
    - supports *iterators* - reference objects with similar properties to pointers
  - Many containers have several *member functions in common*, and *share functionalities* - easy to learn and remember
  - `stack`, `queue` and `priority_queue` are implemented as **Container Adaptors**
    - Container adaptors are *not full container classes*, but classes that provide a *specific interface relying on an object of one of the container classes* (such as `deque` or `list`) to handle the elements
    - The underlying container is encapsulated in such a way that its elements are accessed by the members of the container adaptor independently of the underlying container class used

Data Structures / Containers in C++

Container	Class Template	Remarks
Sequence containers: *Elements are* ordered in a strict sequence *and are* accessed by their position in the sequence		
`array` (C++11)	Array class	1D array of *fixed-size*
`vector`	Vector	1D array of *fixed-size* that can *change in size*
`deque`	Double ended queue	*Dynamically sized*, can be expanded / contracted on *both ends*
`forward_list` (C++11)	Forward list	*Const. time insert / erase* anywhere, done as *singly-linked lists*
`list`	List	*Const. time insert / erase* anywhere, iteration in *both directions*
Container adaptors: *Sequence containers* adapted with specific protocols of access *like LIFO, FIFO, Priority*		
`stack`	LIFO stack	Underlying container is `deque` (default) or as specified
`queue`	FIFO queue	Underlying container is `deque` (default) or as specified
`priority_queue`	Priority queue	Underlying container is `vector` (default) or as specified
Associative containers: *Elements are* referenced by their key *and* not by their absolute position *in the container. They are typically implemented as* binary search trees *and needs the elements to be* comparable		
`set`	Set	Stores *unique elements* in *a specific order*
`multiset`	Multiple-key set	Stores elements in *an order* with *multiple equivalent values*
`map`	Map	Stores *<key, value>* in *an order* with *unique keys*
`multimap`	Multiple-key map	Stores *<key, value>* in *an order* with *multiple equivalent values*
Unordered associative containers: *Elements are* referenced by their key *and* not by their absolute position *in the container. Implemented using a* hash table of keys *and has* fast retrieval *of elements based on keys*		
`unordered_set` (C++11)	Unordered Set	Stores *unique elements* in no particular order
`unordered_multiset` (C++11)	Unordered Multiset	Stores elements in *no order* with *multiple equivalent values*
`unordered_map` (C++11)	Unordered Map	Stores *<key, value>* in *no order* with *unique keys*
`unordered_multimap` (C++11)	Unordered Multimap	Stores *<key, value>* in *no order* with *multiple equivalent values*

Module Summary

- In C, stack needs to be coded by the user and works for a specific type of elements only
- C++ standard library provides ready-made `stack`. It works like a data type
- There are several containers in C++ standard library

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++ projects?
- 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
    - Precompiled 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, Ammendment 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