



Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

- 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?

NPTEL



Tutorial Outline

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

1 Source and Header Files

- Sample C/C++ Files

2 C Preprocessor (CPP): Managing Source Code

• Macros

- Manifest Constants and Macros
- undef
- # & ##

• Conditional Compilation

- #ifdef
- #if
- Use-Cases

• Source File Inclusion

- #include
- #include Guard
- #line, #error
- #pragma
- Standard Macros

3 Tutorial Summary



Source and Header Files

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

NPTEL

Source and Header Files



Source Files

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

- **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 Files* would be there.



Header Files

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

- **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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

- **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:");
    scanf("%d", &n);

    f = fact(n);

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

    return 0;
}
```



Sample Source and Header Files in C

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

- **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

```
// File Solver.h
// User Header files
#ifndef __SOLVER_H // Include Guard. Check
#define __SOLVER_H // Include Guard. define
int quadraticEquationSolver(
    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;
}
```

```
// 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++

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

- **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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

NPTEL

C Preprocessor (CPP): Managing Source Code

Source: [Preprocessor directives, cplusplus.com](https://www.cplusplus.com/doc03/headers/preprocessor/) Accessed 13-Sep-21



C Preprocessor (CPP): Managing Source Code

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

- 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 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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

- 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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

- `#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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

- 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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

- 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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

- 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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

- 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
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#define TABLE_SIZE 50

#else
#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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

- 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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

- 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	With Guard
<pre>// 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; };</pre>	<pre>// 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; };</pre>



C Preprocessor (CPP):

Source File Inclusion: #include Guard

Tutorial T01

Partha Pratim
DasObjectives &
OutlineSource and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary

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

Without Guard	With Guard
<ul style="list-style-type: none"> ○ Class Flight: Needs the info of service provider ○ Class Service: Needs the info of flights it offers <pre>#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 </pre>	<pre>#include<iostream> #include<vector> using namespace std; #ifndef __SERVICE_H #define __SERVICE_H #include "main.h" #include "Flight.h" class Flight; class Service { vector<Flight*> m_Flt; /* ... */ }; #endif // __SERVICE_H #ifndef __FLIGHT_H #define __FLIGHT_H #include "main.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; }; // File main.h // File main.h // File Service.h // File Flight.h // File Flight.h // File main.cpp </pre>



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).

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

Tutorial Summary



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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

`#define`

`undef`

`# & ##`

Conditional

Compilation

`#ifdef`

`#if`

Use-Cases

Source File Inclusion

`#include`

`#include`

Guard

`#line, #error`

`#pragma`

Standard Macros

Tutorial Summary



C Preprocessor (CPP): Predefined Macro Names

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

Macro	Value
<code>__LINE__</code>	Integer value representing the current line in the source code file being compiled
<code>__FILE__</code>	A string literal containing the presumed name of the source file being compiled
<code>__DATE__</code>	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began
<code>__TIME__</code>	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began
<code>__cplusplus</code>	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: <ul style="list-style-type: none">• 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
<code>__STDC_HOSTED__</code>	1 if the implementation is a hosted implementation (with all standard headers available) 0 otherwise

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

`#define`

`undef`

`# & ##`

Conditional

Compilation

`#ifdef`

`#if`

Use-Cases

Source File Inclusion

`#include`

`#include`

Guard

`#line, #error`

`#pragma`

Standard Macros

Tutorial Summary



C Preprocessor (CPP): Predefined Macro Names

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

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

Macro	Value
<code>_STDC_</code>	In C: if defined to 1, the implementation conforms to the C standard. In C++: Implementation defined
<code>_STDC_VERSION_</code>	In C: <ul style="list-style-type: none">199401L: ISO C 1990, Amendment 1199901L: ISO C 1999201112L: ISO C 2011 In C++: Implementation defined
<code>_STDC_MB_MIGHT_NEQ_WC_</code>	1 if multibyte encoding might give a character a different value in character literals
<code>_STDC_ISO_10646_</code>	A value in the form yyyyymmL, specifying the date of the Unicode standard followed by the encoding of wchar_t characters
<code>_STDCPP_STRICT_POINTER_SAFETY_</code>	1 if the implementation has strict pointer safety (see <code>get_pointer_safety</code>)
<code>_STDCPP_THREADS_</code>	1 if the program can have more than one thread

- Macros marked in blue are frequently used



C Preprocessor (CPP): Standard Macro Examples

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

- 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

Tutorial T01

Partha Pratim
Das

Objectives &
Outline

Source and
Header

Sample C/C++ Files

CPP

Macros

#define

undef

&

Conditional

Compilation

#ifdef

#if

Use-Cases

Source File Inclusion

#include

#include

Guard

#line, #error

#pragma

Standard Macros

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

NPTEL



Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

Programming in Modern C++

Tutorial T02: How to build a C/C++ program?: Part 2: Build Pipeline

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 Recap

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

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

NPTEL



Tutorial Objective

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- What is the build pipelines? Especially with reference to GCC
- How to work with C/C++ dialects during build?
- Understanding C/C++ Standard Libraries

NPTEL



Tutorial Outline

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

1 Tutorial Recap

2 Build Pipeline

- Compilers, IDE, and Debuggers
- gcc and g++
- Build with GCC

3 C/C++ Dialects

- C Dialects
- C++ Dialects

4 Standard Library

- C Standard Library
- C++ Standard Library
 - std
 - Header Conventions

5 Tutorial Summary

NPTEL



Build Pipeline

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

NPTEL

Build Pipeline



Build Pipeline

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

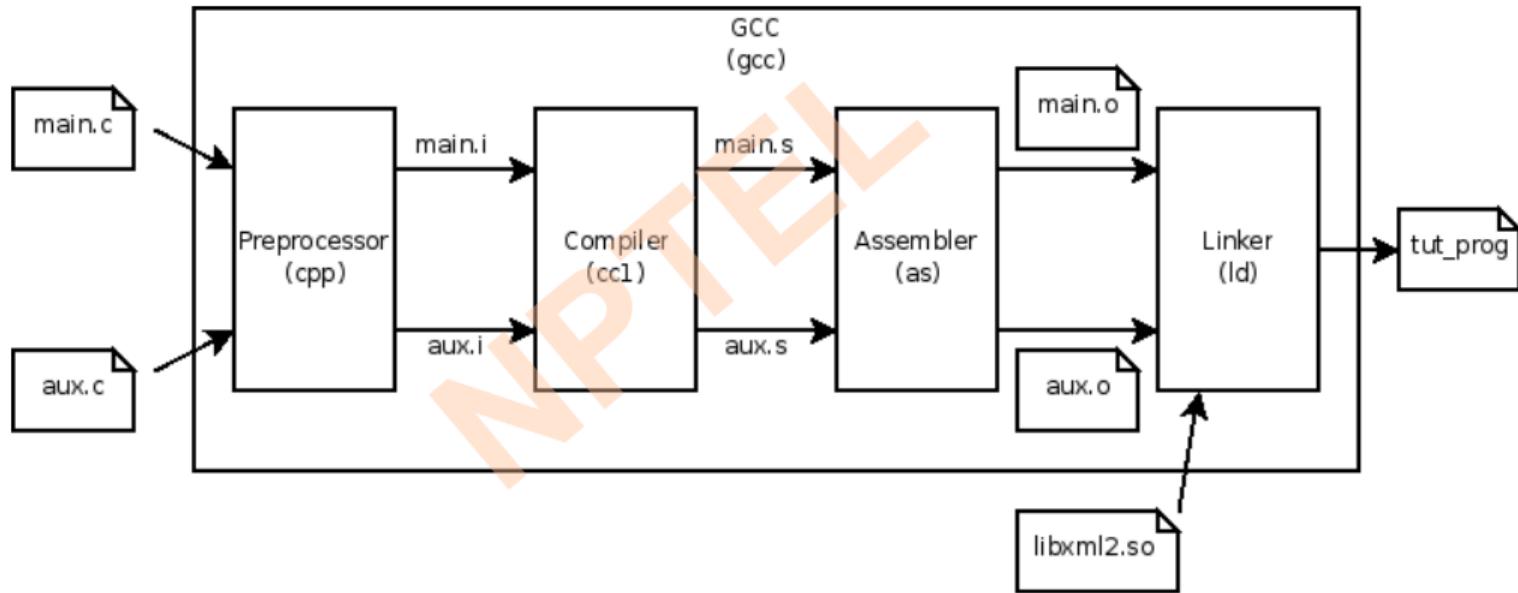
C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary



Source: [GNU Compiler Collection, Wikibooks](#) Accessed 13-Sep-21



Build Pipeline

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- The **C preprocessor (CPP)** has the ability for the inclusion of header files, macro expansions, conditional compilation, and line control. It works on `.c`, `.cpp`, and `.h` files and produces `.i` files
- The **Compiler** translates the pre-processed C/C++ code into assembly language, which is a machine level code in text that contains instructions that manipulate the memory and processor directly. It works on `.i` files and produces `.s` files
- The **Assembler** translates the assembly program to binary machine language or object code. It works on `.s` files and produces `.o` files
- The **Linker** links our program with the pre-compiled libraries for using their functions and generates the executable binary. It works on `.o` (static library), `.so` (shared library or dynamically linked library), and `.a` (library archive) files and produces `a.out` file

File extensions mentioned here are for GCC running on Linux. These may vary on other OSs and for other compilers. Check the respective documentation for details. The build pipeline, however, would be the same.



Compilers

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- The recommended compiler for the course is **GCC**, the **GNU Compiler Collection - GNU Project**. To install it (with **gdb**, the debugger) on your system, follow:
 - **Windows**: [How to install gdb in windows 10](#) on YouTube
 - **Linux**: Usually comes bundled in Linux distribution. Check manual
- You may also use **online versions** for quick tasks
 - **GNU Online Compiler**
 - ▷ From Language Drop-down, choose C ([C99](#)), C++ ([C++11](#)), [C++14](#), [C++17](#), or [C++20](#)
 - ▷ To mark the language for gcc compilation, set `-std=<compiler_tag>`
 - Tags for C are: `ansi`, `c89`, `c90`, `c11`, `c17`, `c18`, etc.
 - Tags for C++ are: `ansi`, `c++98`, `c++03`, `c++11`, `c++14`, `c++17`, `c++20`, etc.
 - Check [Options Controlling C Dialect](#) and [Language Standards Supported by GCC](#)
 - **Code::Blocks** is a free, open source cross-platform IDE that supports multiple compilers including GCC, Clang and Visual C++
 - **Programiz Online Compiler**: [C18](#) and [C++14](#)
 - **OneCompiler**: [C18](#) and [C++17](#)
 - **JDOODLE Online Compiler And Editor**: [C99](#), [C11](#) & [C18](#) and [C++98](#), [C++14](#) & [C++17](#)

- For a compiler, you must know the language version you are compiling for - check to confirm



What is GCC?

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline
Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- **GCC** stands for **GNU Compiler Collections** which is used to compile mainly C and C++ language
- It can also be used to compile Objective C, Objective C++, Fortran, Ada, Go, and D
- The most important option required while compiling a source code file is the name of the source program, rest every argument is optional like a warning, debugging, linking libraries, object file, etc.
- The different options of GCC command allow the user to stop the compilation process at different stages.
- **g++** command is a GNU C++ compiler invocation command, which is used for preprocessing, compilation, assembly and linking of source code to generate an executable file. The different “options” of g++ command allow us to stop this process at the intermediate stage.



What are the differences between gcc and g++?

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

g++	gcc
g++ is used to compile C++ program	gcc is used to compile C program
g++ can compile any .c or .cpp files but they will be treated as C++ files only	gcc can compile any .c or .cpp files but they will be treated as C and C++ respectively
Command to compile C++ program by g++ is: <code>g++ fileName.cpp -o binary</code>	Command to compile C program by gcc is: <code>gcc fileName.c -o binary -lstdc++</code>
Using g++ to link the object files, files automatically links in the std C++ libraries.	gcc does not do this and we need to specify <code>-lstdc++</code> in the command line
g++ compiling .c/.cpp files has a few extra macros	gcc compiling .c files has less predefined macros. gcc compiling .cpp files has a few extra macros
<pre>#define __GXX_WEAK__ 1 #define __cplusplus 1 #define __DEPRECATED 1 #define __GNUG__ 4 #define __EXCEPTIONS 1 #define __private_extern__ extern</pre>	



Build with GCC: Options

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- [1] Place the source (**.c**) and header (**.h**) files in current directory

```
11-09-2021 10:46          157 fact.c
11-09-2021 10:47          124 fact.h
11-09-2021 10:47          263 main.c
```

- [2] Compile source files (**.c**) and generate object (**.o**) files using option “**-c**”. Note additions of files to directory

```
$ gcc -c fact.c
$ gcc -c main.c
```

```
11-09-2021 11:02          670 fact.o
11-09-2021 11:02          1,004 main.o
```

- [3] Link object (**.o**) files and generate executable (**.exe**) file of preferred name (**fact**) using option “**-o**”. Note added file to directory

```
$ gcc fact.o main.o -o fact
```

```
11-09-2021 11:03          42,729 fact.exe
```

- [4] Execute

```
$ fact
Input n
5
fact(5) = 120
```

- [5] We can combine steps [2] and [3] to generate executable directly by compiling and linking source files in one command

```
$ gcc fact.c main.c -o fact
```



Build with GCC: Options

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- [6] We can only compile and generate assembly language (`.s`) file using option “`-S`”

```
$ gcc -S fact.c main.c
```

```
11-09-2021 11:34          519 fact.s
11-09-2021 11:34          1,023 main.s
```

- [7] To stop after prepossessing use option “`-E`”. The output is generated in `stdout` (redirected here to `cppout.c`).

```
$ gcc -E fact.c main.c >cppout.c
```

```
11-09-2021 11:32          21,155 cppout.c
```

Note that CPP:

- Produces a single file containing the source from all `.c` files
- Includes all required header files (like `fact.h`, `stdio.h`) and strips off unnecessary codes present there
- Strips off all comments
- Textually replaces all manifest constants and expands all macros

- [8] We can know the version of the compiler

```
$ gcc --version
```

```
gcc (MinGW.org GCC-6.3.0-1) 6.3.0
```

```
Copyright (C) 2016 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```



Build with GCC: Options

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline
Compilers

gcc and g++
Build with GCC

C/C++ Dialects
C Dialects
C++ Dialects

Standard Library
C Std. Lib.
C++ Std. Lib.
std
Header Conventions

Tutorial Summary

- [9] When we intend to debug our code with `gdb` we need to use “`-g`” option to tell GCC to emit extra information for use by a debugger

```
$ gcc -g fact.c main.c -o fact
```

- [10] We should always compile keeping it clean of all warnings. This can be done by “`-Wall`” flag. For example if we comment out `f = fact(n);` and try to build we get warning, w/o “`-Wall`”, it is silent

```
$ gcc -Wall main.c
```

```
main.c: In function 'main':  
main.c:14:5: warning: 'f' is used uninitialized in this function [-Wuninitialized]  
    printf("fact(%d) = %d\n", n, f);  
    ^~~~~~
```

```
$ gcc main.c
```

With “`-Werror`”, all warnings are treated as errors and no output will be produced



Build with GCC: Options

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

[11] We can trace the commands being used by the compiler using option “`-v`”, that is, verbose mode

```
$ gcc -v fact.c main.c -o fact
```

Using built-in specs.

COLLECT_GCC=gcc

COLLECT_LTO_WRAPPER=c:/mingw/bin/..../libexec/gcc/mingw32/6.3.0/lto-wrapper.exe

Target: mingw32

[truncated]

Thread model: win32

gcc version 6.3.0 (MinGW.org GCC-6.3.0-1)

[truncated]



Build with GCC: Summary of Options and Extensions

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- gcc options and file extensions. Note that `.c` is shown as a placeholder for user provided source files. A detailed list of source file extensions are given in the next point

Option	Behaviour	Input Extension	Output Extension
<code>-c</code>	Compile or assemble the source files, but do not link	<code>.c, .s, .i</code>	<code>.o</code>
<code>-S</code>	Stop after the stage of compilation proper; do not assemble	<code>.c, .i</code>	<code>.s</code>
<code>-E</code>	Stop after the preprocessing stage	<code>.c</code>	To stdout
<code>-o file</code>	Place the primary output in file <code>file</code> (a.out w/o -o)	<code>.c, .s, .i</code>	Default for OS
<code>-v</code>	Print the commands executed to run the stages of compilation	<code>.c, .s, .i</code>	To stdout

- Source file (user provided) extensions

Extension	File Type	Extension	File Type
<code>.c</code>	C source code that must be preprocessed	<code>.cpp, .cc, .cp, .cxx, .CPP, .c++, .C</code>	C++ source code that must be preprocessed
<code>.h</code>	C / C++ header file	<code>.H, .hp, .hxx, .hpp, .HPP, .h++, .tcc</code>	C++ header file
<code>.s</code>	Assembler code	<code>.S, .sx</code>	Assembler code that must be preprocessed

* Varied extensions for C++ happened during its evolution due various adoption practices

* We are going to follow the extensions marked in red

Source: [3.1 Option Summary](#) and [3.2 Options Controlling the Kind of Output](#) Accessed 13-Sep-21



C / C++ Dialects

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers
gcc and g++
Build with GCC

C/C++ Dialects

C Dialects
C++ Dialects

Standard Library

C Std. Lib.
C++ Std. Lib.
std
Header Conventions

Tutorial Summary

NPTEL

C / C++ Dialects



C Dialects

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

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 Dialects: Checking for a dialect

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline
Compilers

gcc and g++
Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- We check the language version (dialect) of C being used by GCC in compilation using the following code

```
/* File Check C Version.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("Unrecognized version of C\n");  
  
    return 0;  
}
```

- We can ask GCC to use a specific dialect by using -std flag and check with the above code for three cases

```
$ gcc -std=c99 "Check C Version.c"  
C99
```

```
$ gcc "Check C Version.c"  
C11
```

```
$ gcc -std=c11 "Check C Version.c"  
C11
```

Default for this gcc is C11



C++ Standards

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

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 Algorithms 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



C++ Dialects: Checking for a dialect

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- We check the language version (dialect) of C++ being used by GCC in compilation using the following code

```
// File Check C++ Version.cpp
#include <iostream>
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 << "Unrecognized version of C++\n";
    return 0;
}
```

- We can ask GCC to use a specific dialect by using -std flag and check with the above code for four cases

```
$ g++ -std=gnu++98 "Check C++ Version.cpp"
C++98
```

```
$ g++ -std=c++11 "Check C++ Version.cpp"
C++11
```

```
$ g++ -std=c++14 "Check C++ Version.cpp"
C++14
```

```
$ g++ "Check C++ Version.cpp"
C++14
```

Default for this g++ is C++14



Standard Library

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

NPTEL

Standard Library



What is Standard Library?

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline
Compilers

gcc and g++
Build with GCC

C/C++ Dialects
C Dialects
C++ Dialects

Standard Library
C Std. Lib.
C++ Std. Lib.
std
Header Conventions

Tutorial Summary

- A *standard library in programming* is the library *made available across implementations of a language*
- These libraries are usually described in *language specifications (C/C++)*; however, they may also be determined (in part or whole) *by informal practices of a language's community (Python)*
- A language's standard library is *often treated as part of the language by its users*, although the *designers may have treated it as a separate entity*
- Many language specifications define a *core set that must be made available in all implementations*, in addition to *other portions which may be optionally implemented*
- The line between a *language and its libraries* therefore *differs from language to language*
- Bjarne Stroustrup, designer of C++, writes:

What ought to be in the standard C++ library? One ideal is for a programmer to be able to find every interesting, significant, and reasonably general class, function, template, etc., in a library. However, the question here is not, "What ought to be in some library?" but "What ought to be in the standard library?" The answer "Everything!" is a reasonable first approximation to an answer to the former question but not the latter. A standard library is something every implementer must supply so that every programmer can rely on it.

- This suggests a *relatively small standard library*, containing only the constructs that "*every programmer*" *might reasonably require when building a large collection of software*
- **This is the philosophy that is used in the C and C++ standard libraries**

Source: [Standard library, Wiki](#) Accessed 13-Sep-21



C Standard Library: Common Library Components

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

Component	Data Types, Manifest Constants, Macros, Functions, ...
<code>stdio.h</code>	Formatted and un-formatted file input and output including functions <ul style="list-style-type: none">• <code>printf</code>, <code>scanf</code>, <code>fprintf</code>, <code>fscanf</code>, <code>sprintf</code>, <code>sscanf</code>, <code>feof</code>, etc.
<code>stdlib.h</code>	Memory allocation, process control, conversions, pseudo-random numbers, searching, sorting <ul style="list-style-type: none">• <code>malloc</code>, <code>free</code>, <code>exit</code>, <code>abort</code>, <code>atoi</code>, <code>strtold</code>, <code>rand</code>, <code>bsearch</code>, <code>qsort</code>, etc.
<code>string.h</code>	Manipulation of C strings and arrays <ul style="list-style-type: none">• <code>strcat</code>, <code>strcpy</code>, <code>strcmp</code>, <code>strlen</code>, <code>strtok</code>, <code>memcp</code>, <code>memmove</code>, etc.
<code>math.h</code>	Common mathematical operations and transformations <ul style="list-style-type: none">• <code>cos</code>, <code>sin</code>, <code>tan</code>, <code>acos</code>, <code>asin</code>, <code>atan</code>, <code>exp</code>, <code>log</code>, <code>pow</code>, <code>sqrt</code>, etc.
<code>errno.h</code>	Macros for reporting and retrieving error conditions through error codes stored in a static memory location called <code>errno</code> <ul style="list-style-type: none">• <code>EDOM</code> (parameter outside a function's domain – <code>sqrt(-1)</code>),• <code>ERANGE</code> (result outside a function's range), or• <code>EILSEQ</code> (an illegal byte sequence), etc.

A header file typically contains manifest constants, macros, necessary struct / union types, `typedef`'s, function prototype, etc.



C Standard Library: math.h

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

```
/* math.h
 * This file has no copyright assigned and is placed in the Public Domain.
 * This file is a part of the mingw-runtime package.
 * Mathematical functions.
 */
#ifndef _MATH_H_
#define _MATH_H_
#ifndef __STRICT_ANSI__ // conditional exclusions for ANSI
// ...
#define M_PI 3.14159265358979323846 // manifest constant for pi
// ...
struct _complex { // struct of _complex type
    double      x;      /* Real part */
    double      y;      /* Imaginary part */
};
__CRTIMP double __cdecl _cabs (struct _complex); // cabs(.) function header
// ...
#endif /* __STRICT_ANSI__ */
// ...
__CRTIMP double __cdecl sqrt (double); // sqrt(.) function header
// ...
#define isfinite(x) ((fpclassify(x) & FP_NAN) == 0) // macro isfinite(.) to check if a number is finite
// ...
#endif /* _MATH_H_ */
```

Source: [C math.h library functions](#) Accessed 13-Sep-21

Programming in Modern C++

Partha Pratim Das

T02.24



C++ Standard Library: Common Library Components

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++
Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

Component	Data Types, Manifest Constants, Macros, Functions, Classes, ...
<code>iostream</code>	Stream input and output for standard I/O <ul style="list-style-type: none">• <code>cout</code>, <code>cin</code>, <code>endl</code>, ..., etc.
<code>string</code>	Manipulation of string objects <ul style="list-style-type: none">• Relational operators, IO operators, Iterators, etc.
<code>memory</code>	High-level memory management <ul style="list-style-type: none">• Pointers: <code>unique_ptr</code>, <code>shared_ptr</code>, <code>weak_ptr</code>, <code>auto_ptr</code>, & <code>allocator</code> etc.
<code>exception</code>	Generic Error Handling <ul style="list-style-type: none">• <code>exception</code>, <code>bad_exception</code>, <code>unexpected_handler</code>, <code>terminate_handler</code>, etc.
<code>stdexcept</code>	Standard Error Handling <ul style="list-style-type: none">• <code>logic_error</code>, <code>invalid_argument</code>, <code>domain_error</code>, <code>length_error</code>, <code>out_of_range</code>, <code>runtime_error</code>, <code>range_error</code>, <code>overflow_error</code>, <code>underflow_error</code>, etc.
Adopted from C Standard Library	
<code>cmath</code>	Common mathematical operations and transformations <ul style="list-style-type: none">• <code>cos</code>, <code>sin</code>, <code>tan</code>, <code>acos</code>, <code>asin</code>, <code>atan</code>, <code>exp</code>, <code>log</code>, <code>pow</code>, <code>sqrt</code>, etc.
<code>cstdlib</code>	Memory alloc., process control, conversions, pseudo-rand nos., searching, sorting <ul style="list-style-type: none">• <code>malloc</code>, <code>free</code>, <code>exit</code>, <code>abort</code>, <code>atoi</code>, <code>strtold</code>, <code>rand</code>, <code>bsearch</code>, <code>qsort</code>, etc.



namespace std for C++ Standard Library

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

C Standard Library

- All names are global
- `stdout`, `stdin`, `printf`, `scanf`

W/o using

```
#include <iostream>

int main() {
    std::cout << "Hello World in C++"
        << std::endl;
    return 0;
}
```

C++ Standard Library

- 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/ using

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

int main() {
    cout << "Hello World in C++"
        << endl;
    return 0;
}
```



Standard Library: C/C++ Header Conventions

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

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



Tutorial Summary

Tutorial T02

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Build Pipeline

Compilers

gcc and g++

Build with GCC

C/C++ Dialects

C Dialects

C++ Dialects

Standard Library

C Std. Lib.

C++ Std. Lib.

std

Header Conventions

Tutorial Summary

- Understood the overall build process for a C/C++ project with specific reference to the build pipeline of GCC
- Understood the management of C/C++ dialects and C/C++ Standard Libraries

NPTEL



Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

Programming in Modern C++

Tutorial T03: How to build a C/C++ program?: Part 3: make Utility

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 Recap

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- Understood the overall build process for a C/C++ project with specific reference to the build pipeline of GCC
- Understood the differences and relationships between source and header files
- Understood how CPP can be harnessed to manage code during build
- Understood the management of C/C++ dialects and C/C++ Standard Libraries



Tutorial Objective

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- Building a software project is a laborious, error-prone, and time consuming process. So it calls for automation by scripting
- **make**, primarily from GNU, is the most popular free and open source dependency-tracking builder tool that all software developers need to know



Tutorial Outline

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build
Why make?

Anatomy of a
makefile

Simple makefile
Variables
Dependency
Source Organization

make Command
Options and Features
Capabilities and
Derivatives

Tutorial Summary

1 Tutorial Recap

2 make Utility

- Example Build
- Why make?

3 Anatomy of a makefile

- Simple makefile
- Simple and Recursive Variables
- Dependency
- Source Organization

4 make Command

- Options and Features
- Capabilities and Derivatives

5 Tutorial Summary



make Utility

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

NPTEL

make Utility

Source: Accessed 15-Sep-21

[GNU make Manual](#)

[GNU Make](#)

[A Simple Makefile Tutorial](#)



make Utility: Example Build

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- Consider a tiny project comprising three files – **main.c**, **hello.c**, and **hello.h** in a folder

main.c	hello.c	hello.h
<pre>#include "hello.h" int main() { // call a function in // another file myHello(); return 0; }</pre>	<pre>#include <stdio.h> #include "hello.h" void myHello(void) { printf("Hello World!\n"); return; }</pre>	<pre>// example include file #ifndef __HEADER_H #define __HEADER_H void myHello(void); #endif // __HEADER_H</pre>

- We build this by executing the command in the current folder (**-I.**):

`gcc -o hello hello.c main.c -I. // Generates hello.o & main.o and removes at the end
which actually expands to:`

```
gcc -c hello.c -I. // Compile and Generate hello.o
gcc -c main.c -I. // Compile and Generate main.o
gcc -o hello hello.o main.o -I. // Link and Generate hello
rm -f hello.o main.o // Is it really necessary? . hello.o & main.o may be retained
```



Why we need make Utility?

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- This manual process of build would be difficult in any practical software project due to:
 - **[Volume]** Projects have *hundreds of folders, source and header files*. They need *hundreds of commands*. It is *time-taking* to type the commands and is *error-prone*
 - **[Workload]** Build needs to be *repeated several times a day* with code changes in some file/s
 - **[Dependency]** Often with the change in one file, all translation units do not need to be re-compiled (assuming that we do not remove `.o` files). For example:
 - ▷ If we change only `hello.c`, we do not need to execute
`gcc -c main.c -I.` // `main.o` is already correct
 - ▷ If we change only `main.c`, we do not need to execute
`gcc -c hello.c -I.` // `hello.o` is already correct
 - ▷ However, if we change `hello.h`, we need to execute all
- There are *dependencies* in build that can be exploited to optimize the build effort
 - **[Diversity]** Finally, we may need to use different build tools for different files, different build flags, different folder structure etc.
- This calls for *automation by scripting*. **GNU Make** is a tool which controls the generation of executables and other non-source files of a program from the program's source files



Why we need make Utility?: What happened in Bell Labs

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

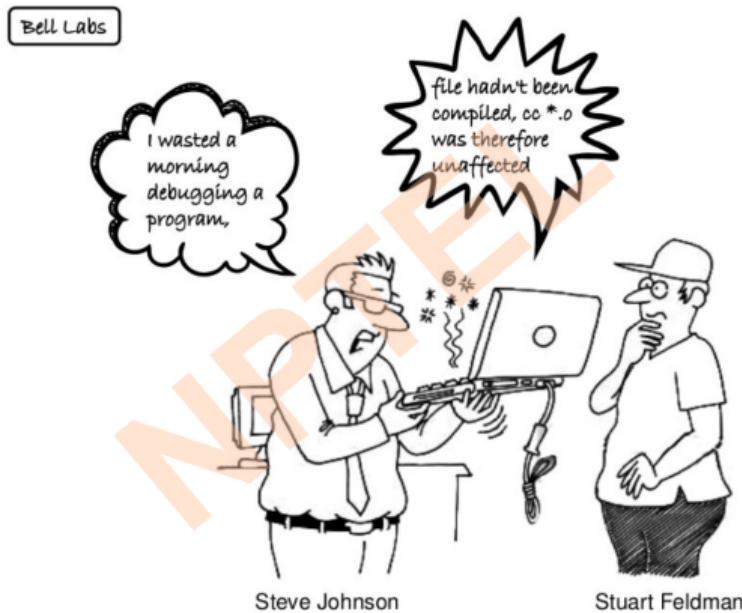
Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary



Broadlinux | Linux of Things

Stuart Feldman created make in April 1976 at Bell Labs

Makefile Martial Arts - Chapter 1. The morning of creation



Anatomy of a makefile

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary



[Make \(software\)](#), Wikipedia

Anatomy of a makefile



makefile: Anatomy

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- A simple make file would be like ([makefile_1.txt](#)):

```
hello: hello.c main.c
      gcc -o hello hello.c main.c -I.
```

Write these lines in a text file named [makefile](#) or [Makefile](#) and run [make](#) command and it will execute the command:

```
$ make
gcc -o hello hello.c main.c -I.
```

- Make file comprises a number of [Rules](#). Every rule has a [target](#) to build, a colon separator (:), zero or more files on which the target depends on and the [commands](#) to build on the next line

```
hello: hello.c main.c # Rule 1
      gcc -o hello hello.c main.c -I.
```

- Note:

- **There must be a tab at the beginning of any command.** Spaces will not work!
- If any of the file in the dependency (hello.c or main.c) change since the last time [make](#) was done ([target hello](#) was built), the rule will fire and the command ([gcc](#)) will execute. This is decided by the last update timestamp of the files
- **Hash (#)** starts a comment that continues till the end of the line



makefile: Anatomy

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

We define reused constants
at the top of the file

```
CC=g++  
CFLAGS=-std=c++11  
  
main:  
    $(CC) -o program main.cc $(CFLAGS)
```

\$ sign followed by parenthesis
indicates to lookup variables

This is a make
rule. If the "all"
rule is not specified,
all rules are executed.
Give any name you like!

Example 1

```
CC=g++  
CFLAGS=-std=c++11  
  
all: test  
  
main:  
    $(CC) -o program main.cc $(CFLAGS)  
  
test:  
    $(CC) -o program test.cc $(CFLAGS)
```

\$ sign followed by parenthesis
indicates to lookup variables

all specifies
which rule
will run by
default.
If you run the
command "make" in
your terminal by default
the test rule will run.

Example 2

How to Write a Makefile with Ease



makefile: Architecture

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build
Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

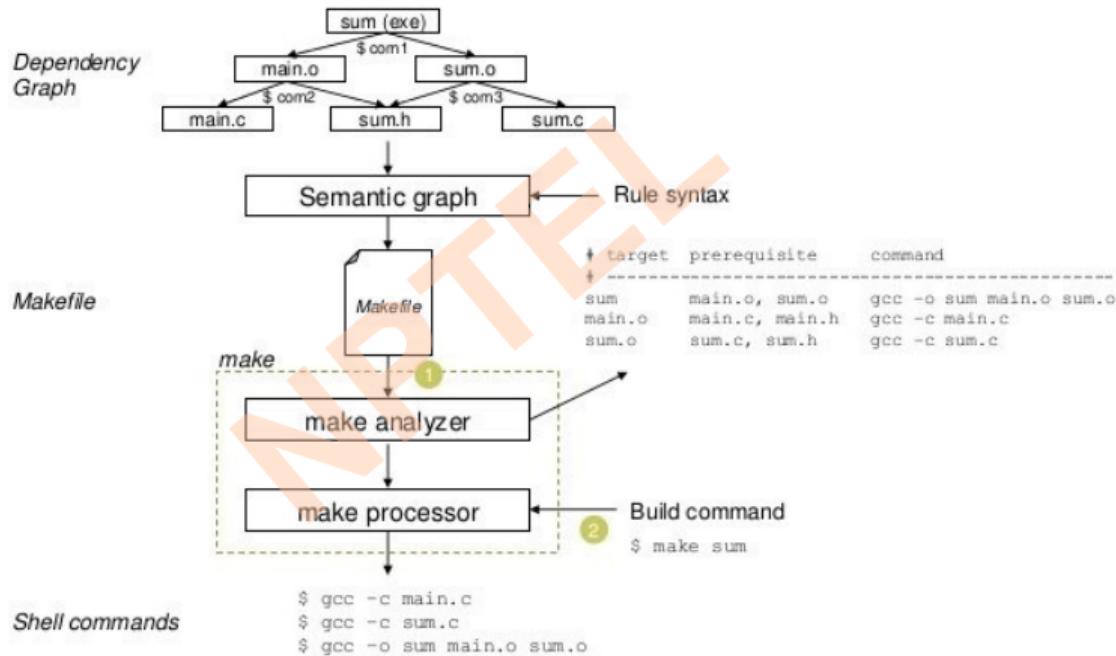
Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary





makefile: Simple and Recursive Variables

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- We can make the make file smarter ([makefile_2.txt](#)) using variables:

```
# CC is a simple variable, gcc is its value
CC := gcc
# CFLAGS is a recursive variable, -I. is its value
CFLAGS = -I.

hello: hello.c main.c      # Rule 1
        $(CC) -o hello hello.c main.c $(CFLAGS)
# $(CC) is value gcc of CC, $CFLAGS is value -I. of CFLAGS, Variables can be expanded by $(.) or ${.}
```

- If there are several commands, to change `gcc` to `g++`, we just need to change one line `CC=g++`.
- There are two types of variables in make ([Chapter 3. Variables and Macros, O'Reilly](#))
 - Simply expanded variables (defined by `:=` operator) and *evaluated as soon as encountered*
 - Recursively expanded variables (by `=`) and *lazily evaluated, may be defined after use*

Simply Expanded	Recursively Expanded
<code>MAKE_DEPEND := \$(CC) -M</code>	<code>MAKE_DEPEND = \$(CC) -M</code>
<code>...</code>	<code>...</code>
<code># Some time later</code>	<code># Some time later</code>

`CC = gcc`

`$(MAKE_DEPEND)` expands to:

<space>-M | gcc -M



makefile: Dependency

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- We are still missing the dependency on the include (header) files. If `hello.h` changes, the above Rule 1 will not detect the need for a re-build. So we improve further (`makefile_3.txt`):

```
CC=gcc
CFLAGS=-I.

#Set of header files on which .c depends
DEPS = hello.h

# Rule 1: Applies to all files ending in the .o suffix
# The .o file depends upon the .c version of the file and the .h files in the DEPS macro
# To generate the .o file, make needs to compile the .c file using the CC macro
# The -c flag says to generate the object file
# The -o $@ says to put the output of the compilation in the file named on the LHS of :
# The $< is the first item in the dependencies list
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

hello: hello.o main.o      # Rule 2: Link .o files
    $(CC) -o hello hello.o main.o -I.
```



makefile: Dependency

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build
Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- We can further simplify on the object files ([makefile_4.txt](#)):

```
CC=gcc
CFLAGS=-I.
DEPS = hello.h
#Set of object files on which executable depends
OBJ = hello.o main.o

# Rule 1: Applies to all files ending in the .o suffix
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $(CFLAGS)

# Rule 2: Linking step, applies to the executable depending on the file in OBJ macro
# The -o $@ says to put the output of the linking in the file named on the LHS of :
# The $^ is the files named on the RHS of :
hello: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS)
```



makefile: Code Organization

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- Finally, let us introduce a source organization that is typical of a large project where under a project **Home** folder, we have the following folders:

- Home**: The make file and the following folders:
 - bin**: The executable of the project. For example `hello / hello.exe`
 - inc**: The include / header (`.h`) files of the project. For example `hello.h`
 - lib**: The local library files (`.a`) of the project
 - obj**: The object files (`.o`) of the project. For example `hello.o & main.o`
 - src**: The source files (`.c/.cpp`) of the project. For example `hello.c & main.c`



makefile: Code Tree

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build
Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

```
Home // Project Home
|
|--- bin // Application binary
|   |
|   |--- hello.exe
|
|--- inc // Headers files to be included in application
|   |
|   |--- hello.h
|
|--- lib // Library files to be linked to application. Check Tutorial Static and Dynamic Library
|
|--- obj // Object files
|   |
|   |--- hello.o
|   |--- main.o
|
|--- src // Source files
|   |
|   |--- hello.c
|   |--- main.c
|
|--- makefile // Makefile
```



makefile: Code Organization

- To handle this hierarchy, we modify as ([makefile_5.txt](#)):

```
CC = gcc
# Folders
BDIR = bin
IDIR = inc
LDIR = lib
ODIR = obj
SDIR = src
# Flags
CFLAGS = -I$(IDIR)
# Macros
_DEPS = hello.h # Add header files here
DEPS = $(patsubst %,$(IDIR)/%, $(DEPS))
_SRC = hello.c main.c # Add source files here
SRC = $(patsubst %,$(SDIR)/%, $(SRC))
_OBJ = hello.o main.o # Add source files here
OBJ = $(patsubst %,$(ODIR)/%, $(OBJ))
# Rule 1: Object files
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS); $(CC) -c -o $@ $< $(CFLAGS) -I.
#Rule 2: Binary File Set binary file here
$(BDIR)/hello: $(OBJ); $(CC) -o $@ $^ $(CFLAGS)
# Rule 3: Remove generated files. .PHONY rule keeps make from doing something with a file named clean
.PHONY: clean
clean: ; del $(ODIR)\*.o $(BDIR)\*.exe
# rm -f $(ODIR)\*.o *~ core $(INCDIR)\*~
```



make Command

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

NPTEL

make Command



make Command: Options and Features

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- The format of `make` command is:

```
make [ -f makefile ] [ options ] ... [ targets ] ...
```

`make` executes commands in the `makefile` to update one or more target `names`

- With no `-f`, `make` looks for `makefile`, and `Makefile`. To use other files do:

```
$ make -f makefile_1.txt          // Using makefile_1.txt
gcc -o hello hello.c main.c -I.
```

- `make` updates a target if its prerequisite files are dated. Starting empty obj & bin folders:

```
$ make -f makefile_5.txt obj/hello.o // Build hello.o, place in obj
gcc -c -o obj/hello.o src/hello.c -Iinc -I.
```

```
$ make -f makefile_5.txt obj/main.o // Build main.o, place in obj
gcc -c -o obj/main.o src/main.c -Iinc -I.
```

```
$ make -f makefile_5.txt bin/hello // Build hello.exe linking .o files and place in bin
gcc -o bin/hello obj/hello.o obj/main.o -Iinc
```

```
$ make -f makefile_5.txt clean      // Remove non-text files generated - obj/*.o & bin/*.exe
del obj\*.o bin\*.exe
```

```
$ make -f makefile_5.txt           // By default targets bin/hello and builds all
gcc -c -o obj/hello.o src/hello.c -Iinc -I.
gcc -c -o obj/main.o src/main.c -Iinc -I.
gcc -o bin/hello obj/hello.o obj/main.o -Iinc
```



make Command: Options and Features

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- More `make` options / features:

- To change to directory `dir` before reading the makefiles, use `-C dir`
- To print debugging information in addition to normal processing, use `-d`
- To specify a directory `dir` to search for included makefiles, use `-I dir`
- To print the version of the `make` program, use `-v`. We are using

`GNU Make 3.81`

`Copyright (C) 2006 Free Software Foundation, Inc.`

`This is free software; see the source for copying conditions.`

`There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.`

`This program built for i386-pc-mingw32`

- `make` can be recursive - one make file may include a command to `make` another
- Multiline:** The backslash ("\`\`") character gives us the ability to use multiple lines when the commands are too long

```
some_file:  
    echo This line is too long, so \  
          it is broken up into multiple lines
```

- Comments:** Lines starting with `#` are used for comments
- Macros:** Besides simple and recursive variables, `make` also supports macros



make Utility: Capabilities and Derivatives

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build
Why make?

Anatomy of a
makefile

Simple makefile
Variables

Dependency
Source Organization

make Command

Options and Features
Capabilities and
Derivatives

Tutorial Summary

- **make** was created by Stuart Feldman in April 1976 at Bell Labs and included in Unix since PWB/UNIX 1.0. He received the 2003 ACM Software System Award for **make**
- **make** is one of the most popular build utilities having the following major **Capabilities**:
 - **make** enables the end user to *build and install a package* without knowing the details of how that is done (it is in the makefile supplied by the user)
 - **make** *figures out automatically which files it needs to update*, based on which source files have changed and *automatically determines the proper order for updating files*
 - **make** is *not limited to any particular language* - C, C++, Java, and so on.
 - **make** is *not limited to building a package* - can control installation/uninstallation etc.
- **make** has several **Derivative** and is available on all OS platforms:
 - **GNU Make** (all types of Unix): Used to build many software systems, including:
 - ▷ GCC, the Linux kernel, Apache OpenOffice, LibreOffice, and Mozilla Firefox
 - **Make for Windows**, GnuWin32 (We are using here)
 - **Microsoft nmake**, a command-line tool, part of Visual Studio
 - **Kati** is Google's replacement of GNU Make, used in Android OS builds. It translates the makefile into **Ninja** (used for Chrome) for faster incremental builds



Tutorial Summary

Tutorial T03

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

make Utility

Example Build

Why make?

Anatomy of a
makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and
Derivatives

Tutorial Summary

- Learnt `make`, the most popular free and open source dependency-tracking builder tool, with its anatomy, architecture and options through a series of examples

NPTEL



Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

Programming in Modern C++

Tutorial T04: How to build a C/C++ program?: Part 4: Static and Dynamic Library

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 Recap

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- Understood the build process and pipeline for C/C++ projects
- Learnt `make` for build automation

NPTEL



Tutorial Objective

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- To understand the role of libraries in C/C++ projects
- To learn about Static and Shared Libraries - how to build and use them

NPTEL



Tutorial Outline

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- 1 Tutorial Recap
- 2 What is a Library?
 - Static vs Shared
- 3 Our Library Project
- 4 Static Library Project
 - Build Steps
 - Execution Trace
- 5 Shared / Dynamic Library Project
 - Build Steps
 - Execution Trace
 - Set Library Path
 - Late Binding and Dynamic Loading
- 6 Dynamic Link Library (DLL) Project: Windows
- 7 Tutorial Summary

NPTEL



What is a Library?

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

What is a Library?

Source: All Accessed 23-Sep-21

[Static library](#), Wikipedia

[Dynamic-link library](#), Wikipedia

[Library \(computing\)](#), Wikipedia

[C standard library](#), Wikipedia

[C++ Standard Library](#), Wikipedia



What is a Library?

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- A library is a package of code that is meant to be reused by many programs. Typically, a C / C++ library comes in two pieces:
 - A *header file* that defines the functionality the library is exposing (offering) to the programs using it. For example, `stdio.h`, `math.h`, etc. in C and `iostream`, `vector`, etc. in C++
 - A *pre-compiled binary* that contains the implementation of that functionality pre-compiled into machine language. For example, `glibc` is the **GNU C (Standard) Library** on Unix

Some libraries may be split into multiple files and/or have multiple header files

- Libraries are pre-compiled because
 - As libraries rarely change, they do not need to be recompiled often. They can just be reused in binary
 - As pre-compiled objects are in machine language, it prevents people from accessing or changing the source code protecting IP

Source: [A.1 — Static and dynamic libraries](#) (Accessed 23-Sep-21)



Types of Library

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

● Static Library

- Consists of routines that are compiled and linked into the program. A program compiled with a static library would have the functionality of the library as a part of the executable
- Extensions:
 - ▷ *Unix*: `.a` (archive)
 - ▷ *Windows*: `.lib`

● Dynamic / Shared Library

- Consists of routines that are loaded into the application at run time
- Extensions:
 - ▷ *Unix*: `.so` (shared object)
 - ▷ *Windows*: `.dll` (dynamic link library)

● Import Library

- An import library automates the process of loading and using a dynamic library
- Extensions:
 - ▷ *Unix*: Shared object (`.so`) file doubles as both a dynamic and an import library
 - ▷ *Windows*: A small static library (`.lib`) of the same name as the dynamic library (`.dll`). The static library is linked into the program at compile time, and then the functionality of the dynamic library can effectively be used as if it were a static library



Static Library

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps
Execution Trace

Shared Library

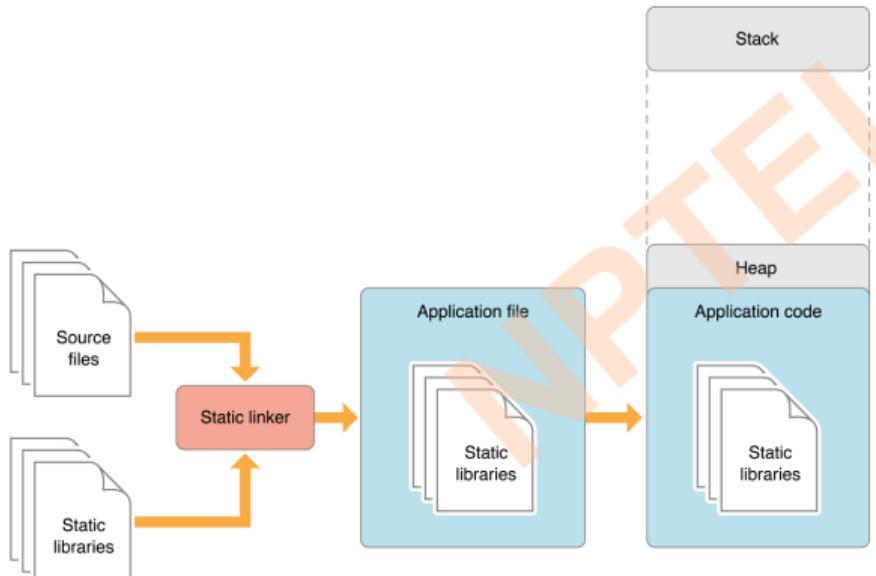
Build Steps
Execution Trace

Set Library Path
Dynamic Loading

Windows DLL

Tutorial Summary

Static Library: Library Code is internal to Application



- Application *needs to recompile* - *difficult version management*
 - If library implementation changes - regular with version upgrade / bug fixes
 - And naturally, if library interface changes - infrequent
- *Large footprint* - especially *bad for mobile apps*
- *Multiple copies* of the same library may be loaded as part of different applications - *bad for mobile apps*
- *Fast in speed* as the library is already loaded and linked



Shared / Dynamic Library

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps
Execution Trace

Shared Library

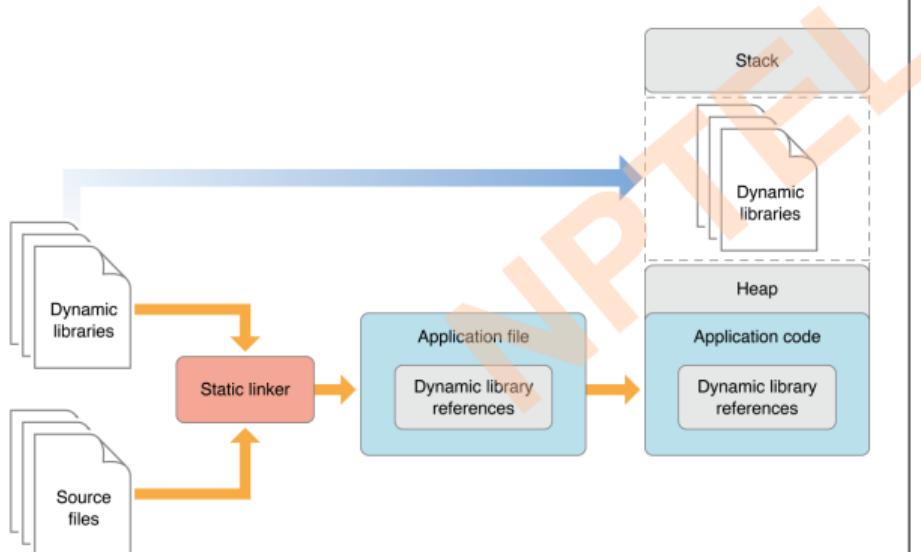
Build Steps
Execution Trace

Set Library Path
Dynamic Loading

Windows DLL

Tutorial Summary

Shared / Dynamic Library: Only Library reference is internal to Application - Library Code is external



- Application *does not need to recompile*
 - *easy version management*
- If library implementation changes - regular with version upgrade / bug fixes
- However, it will need to recompile (like the static library), if library interface changes - infrequent
- *Small footprint* - especially *good for mobile apps*
- *Single copy* of the library will be loaded for different applications - *good for mobile apps*
- The functions in the library needs to be *re-entrant*. Care is needed with static variables
- *Slow in speed* as the library may need to be loaded and linked at run-time



Static vs Shared Library

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

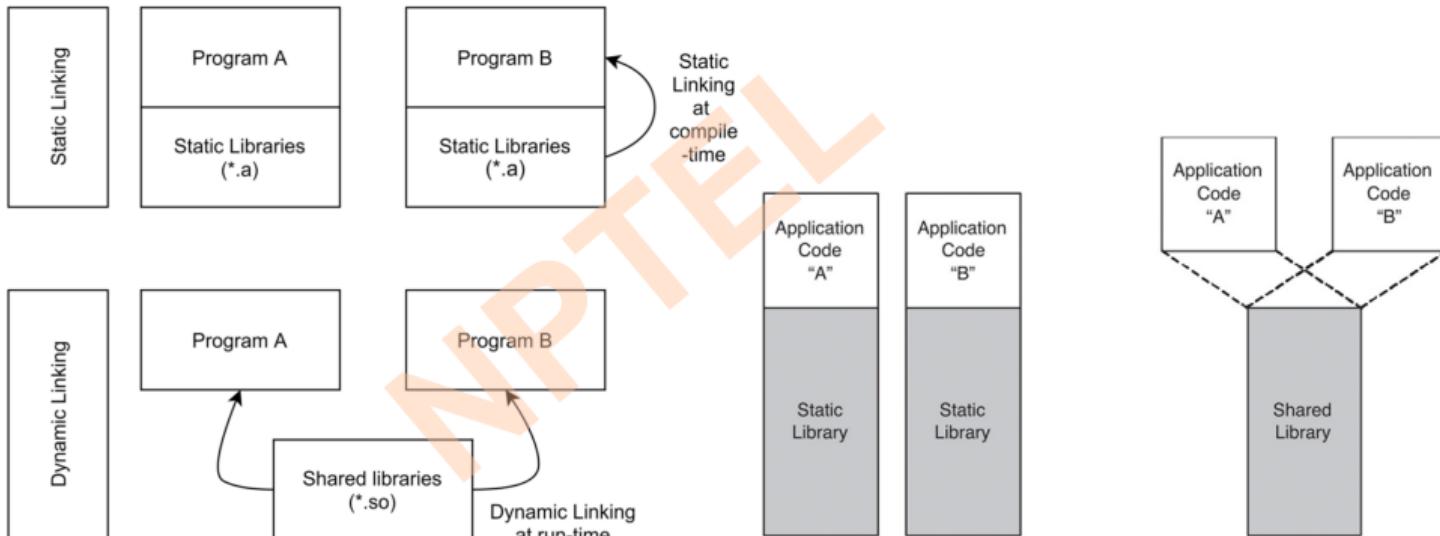
Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary





Static vs Shared Library

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps
Execution Trace

Shared Library

Build Steps
Execution Trace

Set Library Path
Dynamic Loading

Windows DLL

Tutorial Summary

Property	Static Library	Shared Library
<i>Compilation</i>	Recompilation is required for changes in external files	No need to recompile the executable
<i>Linking time</i>	Happens as the last step of the compilation process	Are added during linking when executable file and libraries are added to the memory
<i>Import / Mechanism</i>	Are resolved in a caller at compile-time and copied into a target application by the linker	Get imported at the time of execution of target program by the OS
<i>Size</i>	Are bigger in size, because external programs are built in the executable file	Are smaller, because there is only one copy of shared library that is kept in memory
<i>External file changes</i>	Executable file will have to be recompiled if any changes were applied to external files	No need to recompile the executable - only the shared library is replaced
<i>Time / Performance</i>	Takes longer to execute, as loading into memory happens every time while executing	Faster because shared library is already in the memory
<i>Compatibility</i>	Never has compatibility issue, since all code is in one executable module	Programs are dependent on having a compatible library

Source: [Difference between Static and Shared libraries](#) and [Difference between Static and Shared libraries](#) (Accessed 23-Sep-21)



Our Library Project

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

NPTEL

Our Library Project

Sources: All Accessed 26-Sep-21

Building And Using Static And Shared "C" Libraries

Shared libraries with GCC on Linux

MinGW Static and Dynamic Libraries

Static and Dynamic Libraries in C Language



Our Library Project

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- We present a tiny project to illustrate the ideas of static and shared / dynamic libraries
- We use the same set of header and source files to create and use
 - Static Library
 - Shared / Dynamic Libraryand compare them
- First the projects are created with [gcc](#). These can work on Unix as well as Windows (with [minGW](#): [MinGW - Minimalist GNU for Windows](#))
- Then we show Microsoft-specific process on Windows



Project Files

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

```
/* lib_myMath.h: Header for my mathematical functions */ /* CPP guards omitted for brevity */
int myMax(int, int);
int myMin(int, int);

/* lib_myMath.c: Implementation for my mathematical functions */
#include "lib_myMath.h"
int myMax(int a, int b) { return a>b? a: b; }
int myMin(int a, int b) { return a<b? a: b; }

/* lib_myPrint.h: Header for my printing function */ /* CPP guards omitted for brevity */
void myPrint(const char*, int);

/* lib_myPrint.c: Implementation for my printing function */
#include <stdio.h>
#include "lib_myPrint.h"
void myPrint(const char *name, int a) { printf("%s: %d\n", name, a); }

/* myApp.c: My application */
#include <stdio.h>
#include "lib_myMath.h"
#include "lib_myPrint.h"
int main() {
    myPrint("Max(3, 5)", myMax(3, 5));
    myPrint("Min(3, 5)", myMin(3, 5));
}
```



Project: Folders and Code Organization

```
Home // Library demonstration project with library as well as application Home = Static or Shared
|
|--- app // Application files - will use library headers from ../inc and library from ../lib
|   |--- myApp.c      // Application Source file
|   |--- myApp.exe    // Application Executable
|   |--- myApp.o      // Application Object file
|   |--- myMakeApp.txt // Application Makefile
|
|--- inc // Headers to be included in application and library build
|   |--- lib_myMath.h
|   |--- lib_myPrint.h
|
|--- lib // Library files
|   |
|   |--- obj // Library object files
|   |   |--- lib_myMath.o
|   |   |--- lib_myPrint.o
|   |
|   |--- src // Library source files - will use library headers from ../../inc
|   |   |--- lib_myMath.c
|   |   |--- lib_myPrint.c
|   |
|   |--- lib_mylib.a      // Static Library binary file linked by the application
|   |--- lib_mylib.so     // Shared Library binary file linked by the application
|   |--- myMakeLibrary.txt // Library Makefile
```



Static Library Project

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

NPTEL

Static Library Project

Sources: All Accessed 26-Sep-21

- A.1 — Static and dynamic libraries
- A.2 — Using libraries with Visual Studio
- A.3 — Using libraries with Code::Blocks



Static Library Project: Build Steps

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library
Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- We can build this project by

```
$ gcc lib_myMath.c lib_myPrint.c myApp.c -o myApp
```

- Every time `myApp.c` is updated, we build `lib_myMath.c` and `lib_myPrint.c` even if there is no change. We can avoid the recompile by retaining the object files as:

```
$ gcc -c lib_myMath.c lib_myPrint.c
```

```
$ gcc lib_myMath.o lib_myPrint.o myApp.c -o myApp
```

- When we have many such files that rarely change, we would have a lot of such `.o` files to maintain. These can be bundled into an archive `lib_mylib.a` for ease of reference

```
$ ar rcs lib_mylib.a lib_myMath.o lib_myPrint.o
```

- GNU `ar` utility creates, modifies, and extracts from `archives` (like ZIP) - holding a collection of multiple files in a structure that makes it possible to retrieve the individual files (called `members`)
- Option `rcs` asks to create (`c`) an archive with replacement (`r`) of members and indexing (`s`)
- For details check: [ar\(1\) — Linux manual page](#)

- Finally we use the `.a` file in place of the `.o`'s to link to `myApp.o`

```
$ gcc -o myApp myApp.c lib_myLib.a -L.
```

Alternately, we can place `lib_myLib.a` in default library path and link by `-l_mylib`

```
$ gcc -o myApp myApp.c -l_mylib -L.
```



Static Library Project: Makefiles

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

Static Library makefile

```
# Variables
CC=gcc
AR=ar
SDIR=src
IDIR=../inc
ODIR=obj
CFLAGS=-I$(IDIR)
LFLAGS=-L.
AFLAGS=rcs
# Macros
_DEPS = lib_myMath.h lib_myPrint.h
DEPS = $(patsubst %,$(IDIR)/%,,$(_DEPS))
_SRC = lib_myMath.c lib_myPrint.c
SRC = $(patsubst %,$(SDIR)/%,,$(_SRC))
_OBJ = lib_myMath.o lib_myPrint.o
OBJ = $(patsubst %,$(ODIR)/%,,$(_OBJ))
# Rules
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS) -I.
%.o: $(SDIR)/%.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)
lib_mylib.a: $(OBJ)
    $(AR) $(AFLAGS) -o $@ $^
```

Application makefile

```
# Variables
CC=gcc
IDIR=inc
LDIR=lib
CFLAGS=-I../$(IDIR)
LFLAGS=-L../$(LDIR)
DEPS=
# Rules
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)
myApp: myApp.o
    $(CC) -o myApp myApp.o -l_mylib $(LFLAGS)
```



Static Library Project: Execution Trace

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- Let us build and execute the project

```
// Build Library
D:\Library\Static\lib $ make -f myMakeLibrary.txt
gcc -c -o obj/lib_myMath.o src/lib_myMath.c -I../inc -I.
gcc -c -o obj/lib_myPrint.o src/lib_myPrint.c -I../inc -I.
ar rcs -o lib_mylib.a obj/lib_myMath.o obj/lib_myPrint.o
```

```
// Build Application
D:\Library\Static\app $ make -f myMakeApp.txt
gcc -c -o myApp.o myApp.c -I../inc
gcc -o myApp myApp.o -l_mylib -L../lib
```

```
// Run Application
D:\Library\Static\app $ myApp.exe
Max(3, 5): 5
Min(3, 5): 3
```

- So the static library is working as expected
- Next we check on the contents of various folders



Static Library Project: Directory Listing

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

```
Directory of D:\Library\Static
26-09-2021 14:58    <DIR>          app
26-09-2021 14:58    <DIR>          inc
26-09-2021 14:58    <DIR>          lib

Directory of D:\Library\Static\app
24-09-2021 15:53            179 myApp.c
26-09-2021 11:35        42,348 myApp.exe
26-09-2021 11:35            954 myApp.o
25-09-2021 13:23        215 myMakeApp.txt

Directory of D:\Library\Static\inc
24-09-2021 13:17            66 lib_myMath.h
24-09-2021 13:17            54 lib_myPrint.h

Directory of D:\Library\Static\lib
26-09-2021 11:33        1,722 lib_mylib.a
25-09-2021 13:22        524 myMakeLibrary.txt
26-09-2021 14:58    <DIR>          obj
26-09-2021 14:58    <DIR>          src

Directory of D:\Library\Static\lib\obj
26-09-2021 11:33            716 lib_myMath.o
26-09-2021 11:33            778 lib_myPrint.o

Directory of D:\Library\Static\lib\src
24-09-2021 13:16            143 lib_myMath.c
24-09-2021 13:18            144 lib_myPrint.c
```



Shared / Dynamic Library Project

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps
Execution Trace

Shared Library

Build Steps
Execution Trace
Set Library Path
Dynamic Loading

Windows DLL

Tutorial Summary

NPTEL

Shared / Dynamic Library Project

Sources: All Accessed 26-Sep-21

- A.1 — Static and dynamic libraries
- A.2 — Using libraries with Visual Studio
- A.3 — Using libraries with Code::Blocks



Shared Library Project: Build Steps

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- As in the static case, first we compile `lib_myMath.c` and `lib_myPrint.c` to create the object (`.o`) files using the option `-fPIC`:

```
$ gcc -fPIC -c lib_myMath.c lib_myPrint.c
```

- `-fPIC` stands to mean: Compile for *Position Independent Code* (PIC)
 - For a shared library, the binary of the library and the application are separate and will be separately loaded at run time
 - So when the object files are generated, we need that all jump calls and subroutine calls to use relative addresses, and not absolute addresses
 - `-fPIC` flag tells `gcc` to generate this type of code



Shared Library Project: Build Steps

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- Next step of building the library gets different now as we do not use `ar`. Rather we use `gcc` with the `-shared` option

```
$ gcc -shared -o lib_mylib.so lib_myMath.o lib_myPrint.o
```

- This creates a shared library `lib_mylib.so` where the extension `.so` stands for a *shared object*

- Finally we use the `.so` file to link to `myApp.o`

```
$ gcc -o myApp myApp.c lib_myLib.so -L.
```

- Alternately, we can place `lib_myLib.so` in default library path and link by `-l_mylib`

```
$ gcc -o myApp myApp.c -l_mylib
```



Shared Library Project: Makefiles

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

Shared Library makefile

```
# Variables
CC=gcc
SDIR=src
IDIR=../inc
ODIR=obj
CFLAGS=-I$(IDIR)
LFLAGS=-L.

# Macros
_DEPS = lib_myMath.h lib_myPrint.h
DEPS = $(patsubst %,$(IDIR)/%,,$(_DEPS))
_SRC = lib_myMath.c lib_myPrint.c
SRC = $(patsubst %,$(SDIR)/%,,$(_SRC))
_OBJ = lib_myMath.o lib_myPrint.o
OBJ = $(patsubst %,$(ODIR)/%,,$(_OBJ))

# Rules
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS)
    $(CC) -fPIC -c -o $@ $< $(CFLAGS) -I.
lib_mylib.so: $(OBJ)
    $(CC) -shared -o $@ $^
```

Application makefile

```
# Variables
CC=gcc
IDIR=inc
LDIR=lib
CFLAGS=-I$(IDIR)
LFLAGS=-L$(LDIR)
DEPS=
```

```
# Rules
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)
myApp: myApp.o
    $(CC) -o myApp myApp.o ...$(LDIR)/lib_mylib.so
```



Shared Library Project: Execution Trace

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- Let us build and execute the project

```
// Build Library
D:\Library\Shared\lib $ make -f myMakeLibrary.txt
gcc -fPIC -c -o obj/lib_myMath.o src/lib_myMath.c -I../inc -I.
gcc -fPIC -c -o obj/lib_myPrint.o src/lib_myPrint.c -I../inc -I.
gcc -shared -o lib_mylib.so obj/lib_myMath.o obj/lib_myPrint.o
```

```
// Build Application
D:\Library\Shared\app $ make -f myMakeApp.txt
gcc -c -o myApp.o myApp.c -I../inc
gcc -o myApp myApp.o ../lib/lib_mylib.so
```

```
// Run Application
D:\Library\Shared\app $ myApp.exe
```

Oops! The shared library is not found! The system does not know that lib_mylib.so is in D:\Library\Shared\lib

myApp.exe - System Error



The code execution cannot proceed because lib_mylib.so was not found. Reinstalling the program may fix this problem.

OK



Shared Library Project: Execution Trace

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- If we copy `lib_mylib.so` to the application folder `D:\Library\Shared\app` (where `myApp.exe` resides), the problem goes away and the application runs successfully

```
// Run Application
D:\Library\Shared\app $ myApp.exe
Max(3, 5): 5
Min(3, 5): 3
```

- So the shared library is working as expected
 - However, copying the shared library to the application folder is not preferred as we would need to copy `lib_mylib.so` to every application that would use it.
 - We shall discuss a solution to this library path problem in the next section
- Next we check on the contents of various folders and compare the size of the libraries and applications in static and shared cases

Static Library		Shared Library		Remarks
<code>lib_mylib.a</code>	1,722	<code>lib_mylib.so</code>	27,749	File <code>.so</code> is larger than <code>.a</code> due to the overhead of exported references. With large number of functions in the library (as opposed to just 3) the relative overhead will go down
<code>myApp.exe</code>	42,348	<code>myApp.exe</code>	41,849	Shared <code>.exe</code> would be relatively much smaller with more functions in the library



Shared Library Project: Directory Listing

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

Directory of D:\Library\Shared

```
26-09-2021 14:58    <DIR>        app
26-09-2021 14:58    <DIR>        inc
26-09-2021 14:58    <DIR>        lib
```

Directory of D:\Library\Shared\app

```
24-09-2021 15:53            179 myApp.c
26-09-2021 11:45          41,849 myApp.exe
26-09-2021 11:45           954 myApp.o
26-09-2021 09:41          220 myMakeApp.txt
```

Directory of D:\Library\Shared\inc

```
24-09-2021 13:17           66 lib_myMath.h
24-09-2021 13:17           54 lib_myPrint.h
```

Directory of D:\Library\Shared\lib

```
26-09-2021 11:44         27,749 lib_mylib.so
26-09-2021 10:38          457 myMakeLibrary.txt
26-09-2021 14:58    <DIR>        obj
26-09-2021 14:58    <DIR>        src
```

Directory of D:\Library\Shared\lib\obj

```
26-09-2021 11:44           716 lib_myMath.o
26-09-2021 11:44           778 lib_myPrint.o
```

Directory of D:\Library\Shared\lib\src

```
24-09-2021 13:16           143 lib_myMath.c
24-09-2021 13:18           144 lib_myPrint.c
```



Shared Library Project: Set Library Path

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps
Execution Trace

Shared Library

Build Steps
Execution Trace

Set Library Path
Dynamic Loading

Windows DLL

Tutorial Summary

- We managed to make the application work by copying the shared library to the application folder. However, this is not preferred as we would need to copy `lib_mylib.so` to every application that would use it
- So we need to tell the location for a shared library to the system. This can be done in one of three ways:
 - Place the shared library file in one of the *default library folders* like `/usr/local/lib`, `/usr/local/lib64`, `/usr/lib` and `/usr/lib64` (on Unix) or like `C:\Windows\system32` and `C:\Windows` (on Windows). Refer to the system manual details on these folders
 - Add the folder of the library to the library search folders by *setting path / environment variables*:
 - ▷ Windows: Use `PATH`
 - ▷ Unix: Use `LD_LIBRARY_PATH`
 - We may also *Dynamically Load / Unload* a library at the run-time. This is known as *late binding*. This is achieved by using `dlopen()`, `dlsym()`, and `dlclose()` from `dlfcn.h` header



Shared Library Project: Set Library Path: Windows

- **Windows:** Set environment variables `PATH` to include the folder of the shared library

```
// Check PATH
```

```
D:\Library\Shared\app $ PATH  
PATH=C:\Windows\system32;...
```

```
// Set PATH
```

```
D:\Library\Shared\app $ SET PATH=%PATH%;D:\Library\Shared\lib
```

```
// Check PATH
```

```
D:\Library\Shared\app $ PATH  
PATH=C:\Windows\system32;...;D:\Library\Shared\app
```

```
// Run Application
```

```
D:\Library\Shared\app $ myApp.exe  
Max(3, 5): 5  
Min(3, 5): 3
```

- `PATH` would be reset when we end the command session. Need to set it every time
- We may set the folder in `PATH` Environment Variable - Use (preferred) or System to retain it across sessions. Refer: [How to set the path and environment variables in Windows](#)



Shared Library Project: Set Library Path: Unix

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps
Execution Trace

Shared Library

Build Steps
Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- **Unix:** Set environment variable `LD_LIBRARY_PATH` to include the directory of the shared library as follows:

- For `tcsh` or `csh`:

```
$ setenv LD_LIBRARY_PATH /full/path/to/library/directory:${LD_LIBRARY_PATH}
```

- For `sh`, `bash` and similar:

```
LD_LIBRARY_PATH=/full/path/to/library/directory:${LD_LIBRARY_PATH}  
export LD_LIBRARY_PATH
```

Note:

- `LD_LIBRARY_PATH` is a list of directories in which to search for ELF libraries at execution time
 - The items in the list are separated by either colons or semicolons, and there is no support for escaping either separator
 - A zero-length directory name indicates the current working directory



Shared Library Project: Dynamic Loading

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- We can link shared libraries to a process anytime during its life without automatically loading the shared library by the dynamic loader
- We can do this by using the 'dl' library – load a shared library, reference any of its symbols, call any of its functions, and finally detach it from the process when no longer needed
- This is useful when there may be multiple shared libraries for the same (similar) purpose and we get to know which one to link only at the run-time
- The steps involved are:
 - Load the library from its path using `dlopen()` and get its *handle*
 - Use the *handle* to get access (*function pointers*) to the specific functions we intend to use by `dlsym()`
 - Use the *function pointers* to call the functions in the shared library
 - Finally, unload the library with `dlclose()`
- We first present a simple schematic example



Shared Library Project: Late Binding

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- Known as *Late Binding* as the actual functions to be called are decided only at the run-time

Application	Library
<pre>#include <dlfcn.h> int main() { void* handle = dlopen("hello.so", RTLD_LAZY); // RTLD_LAZY: Relocations shall be performed at an // implementation-defined time, ranging from the // time of the dlopen() call until the first // reference to a given symbol occurs typedef void (*hello_t)(); hello_t myHello = 0; myHello = (hello_t) dlsym(handle, "hello"); myHello(); dlclose(handle); }</pre>	<pre>#include <iostream> using namespace std; extern "C" void hello() { cout << "hello" << endl; }</pre>

- Now we present our project in the context of late binding



Shared Library Project: Dynamic Loading

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

```
/* myDynamicApp.c */
#include <stdio.h>
#include <stdlib.h>
#include "lib_myMath.h"
#include "lib_myPrint.h"
#include <dlfcn.h> /* defines dlopen(), dlsym(), and dlclose() etc. */
int main() { void* lib_handle; /* handle of the opened library */
    const char* error_msg; /* Pointer to an error string */
    /* Load the shared library */
    lib_handle = dlopen("D:\Library\Shared\lib\lib_mylib.so", RTLD_LAZY); /* Library path - dynamic */
    if (!lib_handle) { error_msg = dlerror(); goto Error; }
    /* Define function pointers */
    int (*myMax)(int, int); int (*myMin)(int, int); void (*myPrint)(const char*, int);
    /* Locate the functions in the library. Pick by name and assign to the function pointer */
    myMax = dlsym(lib_handle, "myMax"); if (error_msg = dlerror()) goto Error;
    myMin = dlsym(lib_handle, "myMin"); if (error_msg = dlerror()) goto Error;
    myPrint = dlsym(lib_handle, "myPrint"); if (error_msg = dlerror()) goto Error;
    /* Call the functions */
    (*myPrint)("Max(3, 5)", (*myMax)(3, 5));
    (*myPrint)("Min(3, 5)", (*myMin)(3, 5));
    /* Unload the shared library */
    dlclose(lib_handle);
    return 0; /* Success */
Error: fprintf(stderr, "%s\n", error_msg); exit(1); /* Exit on error */
}
```



Dynamic Link Library (DLL) Project: Windows

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

NPTEL

Dynamic Link Library (DLL) Project: Windows

Sources: All Accessed 27-Sep-21

Microsoft Visual C++ Static and Dynamic Libraries

Walkthrough: Create and use a static library

Walkthrough: Create and use your own Dynamic Link Library (C++)

How to link DLLs to C++ Projects



Microsoft Visual C++ Static and Dynamic Libraries

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- While static and shared library of Unix (specifically **GNU**) is available on Windows through **minGW**, Windows provides Microsoft specific support through MSVC
 - *Static Library*: To work with a static library:
 - ▷ We need to create a static library and an application project (that refers to the library project) using the MSVS IDE
 - Check: [Walkthrough: Create and use a static library](#)
 - ▷ No change in the library or application codes is needed
 - *Dynamic Link Library (DLL)*: To work with a DLL:
 - ▷ We need to create a DLL and an application project (that refers to the DLL project) using the MSVS IDE
 - Check: [Walkthrough: Create and use your own Dynamic Link Library \(C++\)](#)
 - ▷ We need to change the library and / or application codes with `dllexport` and `dllimport` - the Microsoft-specific storage-class attributes for C and C++
 - Check: [dllexport, dllimport](#)
 - ▷ These can be specified by `__declspec()` to export and import functions, data, and objects to or from a DLL
 - ▷ We modify the library project file to illustrate



Library Project Files: Using dllexport

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

```
/* lib_myMath.h: Header for my mathematical functions */ /* File changed for export */
__declspec(dllexport) int myMax(int, int); // storage-class attribute __declspec(dllexport) used
__declspec(dllexport) int myMin(int, int); // to export function from DLL. This is MS-specific

/* lib_myMath.c: Implementation for my mathematical functions */ /* No change */
#include "lib_myMath.h"
int myMax(int a, int b) { return a>b? a: b; }
int myMin(int a, int b) { return a<b? a: b; }

/* lib_myPrint.h: Header for my printing function */ /* File changed for export */
__declspec(dllexport) void myPrint(const char*, int);

/* lib_myPrint.c: Implementation for my printing function */ /* No change */
#include <stdio.h>
#include "lib_myPrint.h"
void myPrint(const char *name, int a) { printf("%s: %d\n", name, a); }

/* myApp.c: My application */ /* No change */
#include <stdio.h>
#include "lib_myMath.h"
#include "lib_myPrint.h"
int main() {
    myPrint("Max(3, 5)", myMax(3, 5));
    myPrint("Min(3, 5)", myMin(3, 5));
}
```



Library Project Files: Using `dllimport`

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps
Execution Trace

Shared Library

Build Steps
Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- In the context of the DLL headers,

```
/* lib_myMath.h */  
__declspec(dllexport) int myMax(int, int);  
__declspec(dllexport) int myMin(int, int);  
  
/* lib_myPrint.h */  
__declspec(dllexport) void myPrint(const char*, int);
```

the application project may include the DLL headers or can just directly import the symbols by `dllimport`

Including DLL headers

```
/* myApp.c: My application */  
#include <stdio.h>  
#include "lib_myMath.h"  
  
#include "lib_myPrint.h"  
  
int main() {  
    myPrint("Max(3, 5)", myMax(3, 5));  
    myPrint("Min(3, 5)", myMin(3, 5));  
}
```

Using `dllexport`

```
/* myApp.c: My application */  
#include <stdio.h>  
#include "lib_myMath.h"  
  
// storage-class attribute __declspec(dllimport) used  
// to import function from DLL. This is MS-specific  
__declspec(dllimport) void myPrint(const char*, int);  
  
int main() {  
    myPrint("Max(3, 5)", myMax(3, 5));  
    myPrint("Min(3, 5)", myMin(3, 5));  
}
```



Tutorial Summary

Tutorial T04

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

What is a
Library?

Static vs Shared

Our Library
Project

Static Library

Build Steps

Execution Trace

Shared Library

Build Steps

Execution Trace

Set Library Path

Dynamic Loading

Windows DLL

Tutorial Summary

- Understood the role of libraries in C/C++ projects in reuse
- Learned about Static and Shared Libraries in Unix and Windows - how to build and use them
- Learned about DLLs

NPTEL



Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

Programming in Modern C++

Tutorial T05: Mixing C and C++ Code: Part 1: Issues and Resolutions

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 Objectives

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- Due to legacy, reuse and several business compulsions, we often need to mix C and C++ codes in the same project
- So we need to learn how to write programs mixing C and C++?

NPTEL



Tutorial Outline

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

1 Mixing C and C++ Codes

- Why Mix C/C++?
- Build all in C++
- Mix C and C++
 - Static Initialization
 - Compiler Compatibility
 - Linkage Issues
 - Exception Issues

2 Common Code Mix Scenarios

- How do I call a C function from C++?
- How do I call a C++ function from C?
- How do I use Pointers to C / C++ Functions?
- How do I include a C Header File?
 - System Header File
 - Non-System Header File

3 Tutorial Summary



Mixing C and C++ Codes

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

NPTEL

Mixing C and C++ Codes

Source: Accessed 16-Sep-21

How to mix C and C++, ISO CPP

Mixing C and C++ Code in the Same Program, Oracle

C++ Core Guidelines: Mixing C with C++

Mixing Code in C, C++, and FORTRAN on Unix



Why do we need to mix C and C++ Codes?

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- Primary reason is **legacy and reuse**. There are possibly trillion lines of well-tested C code available. So reusing them in and / or with C++ needs mixing of codes
- Mixing of codes is actually often needed not only across C and C++, it may be used across C and Python, C# and Python, C++ and Java, and so on to get the **best of both languages** (C/C++ is efficient, C if lightweight for embedded programming, Python has rich libraries and good for web, Java is good for applications with GUI etc.) and be able **to use the available proven libraries**. Actually, we may mix *more than two languages*
- Here are some informative articles on projects using multiple languages:
 - **Polyglot programming – development in multiple languages**, Computer World, 2009
 - **How do you use different coding languages in one program?**, Quora, 2015 and several other in Quora
 - **A Large Scale Study of Multiple Programming Languages and Code Quality**, IEEE, 2016
 - **On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers**, Springer, 2017
- **We restrict the discussions here on mixing C and C++ only**



Why do we need to mix C and C++ Codes?

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **There are two typical situations:**

- *Both C header and C source files are available and editable:* An existing project that needs to be migrated to C++ in full or parts (needs to be reused). Two options:
 - ▷ **Mix C and C++ codes:** Compile C code with C compiler, and C++ code with C++ compiler and then link by C++ ([how?](#))
 - ▷ **Build all in C++:** Build both C and C++ codes with C++ compiler and link
- *Only C header files are available:* For third party library where source is already pre-compiled. In fact, *the header file too may not be editable*. For example,
 - ▷ To write a C++ program/library that does scientific calculations, we would possibly use GSL ([GNU Scientific Library](#)), and it is written in C
 - ▷ The C game codes called from the C++ graphics engine

We would like to wrap all the C functions to use in nice C++ style functions, perhaps with the necessary exceptions and returning a `std::string` instead of having to pass a `char*` buffer as argument. Now we have only one option:

- ▷ **Mix C and C++ codes:** Compile C code with C compiler, and C++ code with C++ compiler and then link by C++ ([how?](#))



Build C and C++ Codes as C++

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **Build as C++:** In a mixed code project where all header and source files are *available and editable*, we can **compile all the code** (even the C-style code) using a **C++ compiler**. For example, using **g++** for both **.c** and **.cpp** files. That eliminates the need to mix C and C++.
- However, it is not easy to build the C code by C++ compiler unless the C code strictly uses the common subset of C and C++ (Check the Tutorial on *Compatibility of C and C++* for details). For example, consider the simple C program below where difference of behavior in C and C++ compilers are marked in different colors:

```
/* cStyle.c */  
#include <stdio.h>  
int main() {  
    double sq2=sqrt(2); // (1): math.h missing. Warning in C89. Error in C++98  
    printf("sizeof('a'): %d",  
          sizeof('a'));// (2): 'a' is int in C, char in C++. Outputs 4 in C89. Outputs 1 in C++98  
    char c;  
    void* pv = &c;  
    int* pi = pv; // (3): Implicit conversion from void* to int*. Okay in C89. Error in C++98  
    int class = 5; // (4): class is a reserved word in C++. Okay in C89. Error in C++98  
}
```

- So the **C code needs to be ported for C++**



Build C and C++ Codes as C++

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **Build as C++:** While building a C/C++ project (both C and C++ codes) with C++ is preferable from language perspective it has a number of shortcomings from engineering viewpoint
 - The *C-style code may need porting* as the *C++ compiler is more strict* - as we have seen in the example
 - *Porting may involve substantial cost* in terms of developer effort as well as project time. This may *not be affordable from the business perspectives*
 - With porting we also need to *create new testplan for C++*, perform *extensive testing* to match the regression. This involves further cost in terms of tester effort as well as project time. This may *not be affordable from the business perspectives*
 - If we are porting a stable C code, even after regression clean and testing, it is *likely to break some of the existing functionality* in the software or even in customer's code if the C code is part of a library provided to the customer. This too may *not be affordable from the business perspectives*
 - So building as C++ is *feasible only in some select situations* though it is *preferred*



Mixing C and C++ Codes: Issues and Remedies

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++
Why Mix C/C++?

Build all in C++
Mix C & C++
Static Initialization

Compiler
Compatibility
Linkage Issues
Exception Issues

Common Mix
C from C++
C++ from C
Pointers to Functions
C Header File
System
Non-System
Tutorial Summary

- When we mix codes, that is, compile C code by C Compiler and C++ code by C++ Compiler - all into `.o` files - we expect to be free from language-specific issues in *individual translation units*
- However, issues arise as we work with different versions of compilers, link the `.o` files of translation units, and as control flows across units during execution, and so on:
 - Static Initialization Issues*: While compiling `main()`, static initialization is handled differently in C and C++
 - Compiler Incompatibility Issues*: The compilers may have *incompatibility in calling conventions, definitions of basic types* (like `int`, pointer), *runtime library*, etc.
 - C Library Incompatibility Issues*: If the C++ compiler provides its own versions of the C headers, the headers used by the C compiler must be compatible
 - Linkage Issues*: C and C++ linkage conventions differ
 - Exception Issues*: C and C++ use drastically different exception models
 - Scope of struct Issue*: Scoping of *nested struct* differ between C and C++. Check the Tutorial on *Compatibility of C and C++* for details



Mixing C and C++ Codes: Issues and Remedies

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **Static Initialization Issues:** In C and C++ both the `static` variables are constructed and initialized before `main()`. But they have different semantics and handling for `static`
 - In C, a `static` initializer must be a constant
 - In C++, a `static` variable must be constructed – its constructor must get called

So the following code compiles in C++, but fails in C

```
#include <stdio.h>
int init(void) { return 10; }

static int i = init(); /* Error in C: initializer element is not constant. Okay in C++ */

int main() { printf("i = %d", i); }
```

Hence,

- C++ compiler generates an additional `Start` function, where all *global function calls* (including constructors) are executed before `main` starts
- C compiler does not generate such `Start` function, `main` starts as soon as it is loaded

So,

- **RULE 1: Use C++ compiler when compiling main()**



Mixing C and C++ Codes: Issues and Remedies

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **Compiler and C Library Incompatibility Issues:** To alleviate the problems outlined, we should

- Use compilers (preferably) from the same vendor (say, `gcc`)
- Have same / compatible versions (for example, use the same calling conventions, define basic types such as `int`, `float` or pointer in the same way)
- C runtime library must also be compatible with the C++ compiler
- If the C++ compiler provides its own versions of the C headers, the versions of those headers used by the C compiler must be compatible

So,

- **RULE 2: C and C++ compilers must be compatible**



Mixing C and C++ Codes: Issues and Remedies

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++
Why Mix C/C++?

Build all in C++

Mix C & C++
Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **Linkage Issues:** C and C++ linkage conventions differ

- In C

- ▷ Every function is in global scope
 - ▷ Every function name is unique

Hence the **C function name** is used by the linker to identify the function across units

- In C++

- ▷ A function may be in multiple scopes
 - Global Scope
 - Non-Static Member Function in class Scope
 - Static Member Function in class Scope
 - namespace Scope
 - ▷ A function may be overloaded in any of these scopes

Hence the C++ function name is not unique. The linker, therefore, uses the combination of the **C++ function name** and **signature (parameter types)** to create unique identity, called **mangled name**



Mixing C and C++ Codes: Issues and Remedies

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **Linkage Issues:** Consider C++ code with 2 overloads of `print()` in multiple scopes

```
#include <iostream>
using namespace std;
void print(int) { cout << "int" << endl; } // Global
void print(double) { cout << "double" << endl; }
class MyClass { int i; double d; const char *pc; public:
    MyClass(int _i = 1, double _d = 1.1, const char *_pc = "Hello") : i(_i), d(_d), pc(_pc) { }
    void print(int) { cout << "MyClass int " << i << endl; } // Class member
    void print(double) { cout << "MyClass double " << d << endl; }
};
class MyOtherClass { public:
    static void print(int) { cout << "MyOtherClass int " << endl; } // Class static member
    static void print(double) { cout << "MyOtherClass double " << endl; }
};
namespace MySpace {
    void print(int) { cout << "MySpace int" << endl; } // namespace member
    void print(double) { cout << "MySpace double" << endl; }
}
int main() { MyClass a;
    print(10); print(10.10);
    a.print(10); a.print(10.10);
    MyOtherClass::print(10); MyOtherClass::print(10.10);
    MySpace::print(10); MySpace::print(10.10);
}
```



Mixing C and C++ Codes: Issues and Remedies

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **Linkage Issues:** The mangled and un-mangled names of functions are:

Function	gcc 6.3.0	msvc 18.00	Mangled?
<i>// Global: Overloaded</i>			
print(int)	_Z5printi	?print@@YAXH@Z	Yes
print(double)	_Z5printd	?print@@YAXN@Z	Yes
<i>// Class member: Overloaded</i>			
MyClass::print(int)	_ZN7MyClass5printEi	?print@MyClass@@QAEXH@Z	Yes
MyClass::print(double)	_ZN7MyClass5printEd	?print@MyClass@@QAEXN@Z	Yes
<i>// Class static member: Overloaded</i>			
MyOtherClass::print(int)	_ZN12MyOtherClass5printEi	?print@MyOtherClass@@SAXH@Z	Yes
MyOtherClass::print(double)	_ZN12MyOtherClass5printEd	?print@MyOtherClass@@SAXN@Z	Yes
<i>// namespace member: Overloaded</i>			
MySpace::print(int)	_ZN7MySpace5printEi	?print@MySpace@@YAXH@Z	Yes
MySpace::print(double)	_ZN7MySpace5printEd	?print@MySpace@@YAXN@Z	Yes
<i>// Global: Not Overloaded</i>			
main()	_main	_main	No

Therefore C and C++ compilers need to handle the names differently. *As C compiler does not know about mangling, we need to tell the C++ compiler not to mangle the names in the C context*



Mixing C and C++ Codes: Issues and Remedies

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **Linkage Issues:** The `extern "C"` linkage specifier can prevent the C++ compiler from mangling the names. Declaring a function within `extern "C"` in the code, we can call a C function from C++, or a C++ function from C. We may use `extern "C"`

- for each function

```
extern "C" void foo(int);
```

- for each function in a scope

```
extern "C" {  
    void foo(int);  
    double bar(double);  
};
```

- or for the entire header file by using include guards

```
#ifdef __cplusplus  
extern "C" {  
#endif  
    void foo(int);  
    double bar(double);  
    ...  
#ifdef __cplusplus  
}  
#endif
```

Note that the macro `__cplusplus` is defined when the C++ compiler is used. Hence, `extern "C"` is not exposed to C compiler (it does not recognize it)



Mixing C and C++ Codes: Issues and Remedies

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- **Linkage Issues:** As we have seen, using `extern "C"` within `__cplusplus` guard, we can make a header that works appropriately for C and C++ both. For an illustrative example, you may refer to the video: [C Programming — Advance Topic And Mixing C with C++ Code, YouTube](#) (Accessed 19-Sep-21)

We now have the next rule:

- **RULE 3: C++ compiler should direct the linking process (so it can get its special libraries)**



- **Exception Issues:** C and C++ use different exception models

- **Propagating Exceptions**

- ▷ What happens if a *C++ function is called from a C function, and the C++ function throws an exception?* The C++ standard is somewhat vague this
 - ▷ In some systems like **Oracle Developer Studio C++** this will work properly. In it *if a C function is active when a C++ exception is thrown, the C function is passed over in the process of handling the exception*

- **Mixing Exceptions with `setjmp` and `longjmp`**

- ▷ The C++ exception mechanism and C++ rules about *destroying objects that go out of scope* are likely to be *violated by a longjmp, with unpredictable results*
 - ▷ Some C++ experts believe that `longjmp` should not be integrated with exceptions, due to the difficulty of specifying exactly how it should behave.
 - ▷ If we must use `longjmp` in C code that we are mixing with C++, we need to *ensure that a longjmp does not cross over an active C++ function*



Common Code Mix Scenarios

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

NPTEL

Common Code Mix Scenarios



Common Code Mix Scenarios

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- How do I call a C function from C++?
- How do I call a C++ function from C?
 - Non-Member
 - ▷ Global / namespace
 - ▷ static
 - Member
 - ▷ Non-static
 - Non-virtual
 - virtual
 - ▷ static
 - Overloaded

- How do I include a C Header File?
 - System / Standard Library Headers
 - Non-System / User-defined Headers
 - ▷ Editable Headers
 - ▷ Non-Editable Headers
- How do I use Pointers to C / C++ Functions?
- How do I manipulate with objects in a C / C++ mix project?



How do I call a C function from C++?

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- Declare the C function `extern "C"` (in the C++ code) and call it (from the C or C++ code):

```
// C++ code
extern "C" void f(int);    // one way
extern "C" {
    int g(double);        // another way
    double h();
};

void code(int i, double d) {
    f(i);
    int ii = g(d);
    double dd = h();
    // ...
}
```

```
/* C code: */
void f(int i) {
    /* ... */
}
int g(double d) {
    /* ... */
}
double h() {
    /* ... */
}
```

Note that C code does not need to be edited
So pre-compiled .o file may also be used

- Note that C++ type rules, not C rules, are used. So a function declared `extern "C"` cannot be called with the wrong number of arguments. For example:

```
// C++ code
void more_code(int i, double d) {
    double dd = h(i,d); // error: unexpected arguments
    // ...
}
```



How do I call a C++ function from C?: Non-Member and Member Functions

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++
Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- Declare the C+ `extern "C"` (in the C++ code) and call it (from the C or C++ code):

```
// C++ code
extern "C" void f(int);
void f(int i) {
    // ...
}
// This works only for non-member functions
```

```
/* C code: */
void f(int);
void cc(int i) {
    f(i);
    /* ... */
}
```

- To call member functions (including virtual functions) from C, a simple wrapper needs to be provided. For example:

```
// C++ code
class C {
    // ...
    virtual double f(int);
};
// wrapper function
extern "C"
double call_C_f(C* p, int i) {
    return p->f(i);
}
```

```
/* C code: */
double call_C_f(struct C* p, int i);
void ccc(struct C* p, int i) {
    double d = call_C_f(p,i);
    /* ... */
}
```



How do I call a C++ function from C?: Overloaded Functions

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- To call overloaded functions from C, wrappers, with distinct names for the C code to use, must be provided. For example:

```
// C++ code
void f(int);
void f(double);
extern "C" void f_i(int i) { f(i); }
extern "C" void f_d(double d) { f(d); }
```

```
/* C code: */
void f_i(int);
void f_d(double);
void cccc(int i,double d) {
    f_i(i);
    f_d(d);
    /* ... */
}
```

Note that these techniques can be used to call a C++ library from C code even if it is not desirable or possible to modify the C++ headers



How do I use Pointers to C / C++ Functions?

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- A pointer to a function must specify whether it points to a C function or to a C++ function:

```
typedef int (*pfun)(int); // pfun points to a C++ function
extern "C" void foo(pfun); // foo is a C function taking a pointer to a C++ function
extern "C" int g(int)      // g is a C function
...
// Type Mismatch Error!
foo(g);                  // foo called with a pointer to g, a C function
```

- To match the linkage of a pointer-to-function with the functions to which it will point, all declarations in this example needs to be inside `extern "C"` brackets, ensuring that the types match

```
extern "C" {
    typedef int (*pfun)(int);
    void foo(pfun);
    int g(int);
}
...
foo(g); // Types Match. Now OK
```



How can I include a standard C header file in my C++ code?

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- #including a standard header file such as `<cstdio>`, is nothing unusual. For example:

// C++ code

```
// Nothing unusual in #include line
#include <cstdio>
int main() {
    // Nothing unusual in the call either
    std::printf("Hello world");
    // ...
}
```

/* C code */

```
/* Compiled using a C++ compiler */
/* Nothing unusual in #include line */
#include <stdio.h>
int main() {
    /* Nothing unusual in the call either */
    printf("Hello world");
    // ...
}
```

- The `std::` part of the `std::printf()` is for the `std` namespace where C++ Standard Library components like `<cstdio>` belongs
- So a C code using the C++ compiler can still use just `printf()` from `<stdio.h>` as C Standard Library belongs to global
- Care is needed if there is difference between the C header and its corresponding C++ version



How can I include a standard C header file in my C++ code?

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++
Why Mix C/C++?

Build all in C++
Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- To summarize

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



How can I include a non-system C header file in my C++ code?: Editable Case

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- *If the C header is editable*, add the `extern "C" {...}` logic inside the header and guard it by `__cplusplus` to let C compiler to ignore it

```
#ifdef __cplusplus /* C++ compiler notes, C compiler ignores */
extern "C" {
#endif

void f(int i, char c, float x) /* Original Code of the Header */

#ifdef __cplusplus
}
#endif
```

Now the C header may be `#included` without `extern "C"` in the C++ code:

```
// C++ code
// Get declaration for f(int i, char c, float x)
#include "my-C-code.h" // Note: nothing unusual in #include line
int main() {
    f(7, 'x', 3.14); // Note: nothing unusual in the call
    // ...
}
```



How can I include a non-system C header file in my C++ code?: Non-Editable Case

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++

Why Mix C/C++?

Build all in C++

Mix C & C++

Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- *If the C header is not editable*, the `#include` line must be wrapped in an `extern "C" { /*...*/ }` construct. This tells the C++ compiler that the functions declared in the header file are C functions

```
// C++ code
extern "C" {
    // Get declaration for f(int i, char c, float x)
    #include "my-C-code.h"
}

int main() {
    f(7, 'x', 3.14); // Note: nothing unusual in the call
    // ...
}
```



Tutorial Summary

Tutorial T05

Partha Pratim
Das

Objectives &
Outline

Mixing C & C++
Why Mix C/C++?

Build all in C++

Mix C & C++
Static Initialization

Compiler
Compatibility

Linkage Issues

Exception Issues

Common Mix

C from C++

C++ from C

Pointers to Functions

C Header File

System

Non-System

Tutorial Summary

- We have learnt why it is often necessary to mix C and C++ codes in the same project
- We have explored the basic issues of mixing and learnt the ground rules
- In addition to the rules, we have three mechanisms to ease code mixing
 - Use `extern "C"` in C++ for all functions to be called from both C and C++
 - Guard `extern "C"` with `__cplusplus` guard for use with C
 - Provide `wrappers` for C++ data members, member functions, and overloaded functions for use with C



Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

Programming in Modern C++

Tutorial T06: Mixing C and C++ Code: Part 2: Project Example

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 Recap

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++
Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

- We have learnt why it is often necessary to mix C and C++ codes in the same project
- We have explored the basic issues of mixing and learnt the ground rules
- In addition to the rules, we have three mechanisms to ease code mixing
 - Use `extern "C"` in C++ for all functions to be called from both C and C++
 - Guard `extern "C"` with `__cplusplus` guard for use with C
 - Provide `wrappers` for C++ data members, member functions, and overloaded functions for use with C



Tutorial Objectives

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

- Walk through a C / C++ mix project using the rules and scenarios of mixing

NPTEL



Tutorial Outline

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

1 Tutorial Recap

2 Mixing C and C++ Codes

- Rules
- Common Code Mix Scenarios

3 How do I manipulate with objects in a C / C++ mix project?

- Data.h
- Data.cpp
- Data_Wrap.cpp
- App.c
- main.cpp
- makefile
- Execution
- Call Trace

4 Advanced Code Mix Scenarios and Advisory

5 Tutorial Summary



Mixing C and C++ Codes

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

NPTEL

Mixing C and C++ Codes

Source: Accessed 16-Sep-21

How to mix C and C++, ISO CPP

Mixing C and C++ Code in the Same Program, Oracle

C++ Core Guidelines: Mixing C with C++

Mixing Code in C, C++, and FORTRAN on Unix



Mixing C and C++ Codes: Rules: Recap

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

- **RULE 1: Use C++ compiler when compiling main()**
- **RULE 2: C and C++ compilers must be compatible**
- **RULE 3: C++ compiler should direct the linking process**

NPTEL



Mixing C and C++ Codes: Scenarios: Recap

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

- How do I call a C function from C++?

```
extern "C" { void f(int); }
```

- How do I call a C++ function from C?

- Non-Member

```
extern "C" { void f(int); }
```

- Member

```
class C { /*...*/  
    virtual double f(int);  
};  
// wrapper function  
extern "C" double call_C_f(C* p, int i)  
{ return p->f(i); }
```

- Overloaded

```
void f(int);  
void f(double);  
// wrapper functions  
extern "C"  
{  
    void f_i(int i) { f(i); }  
    void f_d(double d) { f(d); }  
}
```

- How do I include a C Header File?

- System / Standard Library Headers
 - Non-System Headers: Editable

```
#ifdef __cplusplus /* C compilers skip */  
extern "C" {  
#endif  
/* Original Code of the Header */  
#ifdef __cplusplus  
}  
#endif
```

- Non-System Headers: Non-Editable

```
// In C++ header / source  
extern "C" {  
    #include "my-C-code.h" // C Header  
}
```

- How do I use Pointers to C / C++ Functions?

```
extern "C" {  
    typedef int (*pfun)(int);  
    void foo(pfun);  
    int g(int); // foo(g) is valid  
}
```



C / C++ Mixed Project

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

NPTEL

C / C++ Mixed Project



C / C++ Mixed Project: Manipulating with Objects from C

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++
Rules

Scenarios

C / C++ Mixed
Project

Data.h
Data.cpp

Data_Wrap.cpp

App.c
main.cpp

makefile
Execution

Call Trace

Advanced Mix

Tutorial Summary

- We present an example project comprising the following files to summarize various code mixing scenarios in an integrated manner:
 - **Data.h**: C/C++ common header containing:
 - ▷ definition of `class Data`
 - ▷ prototypes of C functions to interact with `class Data`, and
 - ▷ prototypes of C++ wrappers providing access points for C to call member functions in `class Data`
 - **Data.cpp**: Implementations of `class Data`
 - **Data_Wrap.cpp**: Implementations of C++ wrapper functions for `class Data`
 - **App.c**: Implementations of C functions for interacting with `class Data`
 - **main.cpp**: `main` to invoke the C functions
 - **makefile**: Mix build of C and C++, and C++ link script



C / C++ Mix Project: Mix Scenarios

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h
Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

- Calling C functions from C++ ([main.cpp](#))

- `Data* c_create_object(int); /* C function to create an object */`
 - `void c_access_object(Data*); /* C function to access an object */`
 - `void c_release_object(Data*); /* C function to release an object */`

- Calling C++ functions from C ([App.c](#))

- `Data* call_create(int); /* C++ wrapper to create an object by new */`
 - `int call_get(Data*); /* C++ wrapper to get the state of an object by get */`
 - `void call_set(Data*, int); /* C++ wrapper to change the state of an object by set */`
 - `void call_release(Data*); /* C++ wrapper to release an object by delete */`

- Passing an object from C to C++ ([main.cpp](#))

- `Data* c_create_object(int); /* C function to create an object */`

- Passing an object from C++ to C ([main.cpp](#))

- `void c_access_object(Data*); /* C function to access an object */`
 - `void c_release_object(Data*); /* C function to release an object */`

- C++ wrappers for object creations, get / set, and release ([Data_Wrap.cpp](#))

- C functions for object creations, get / set, and release ([App.c](#))

- Common header for C and C++ ([Data.h](#))

- `typedef struct Data Data; /* Incomplete Type to access Data* in C function */`



C / C++ Mix Project: Data.h

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

```
/* C Header and C++ Header Data.h - can be read by both C and C++ compilers */
#ifndef __DATA_H      /* include Guard */
#define __DATA_H
#ifndef __cplusplus /* Guard for C++ */
    class Data { int d_;
    public: Data(int=0); ~Data();
        int get(); void set(int);
    };
#else                /* Guard for C */
    typedef struct Data Data;
#endif
#ifndef __cplusplus /* Guard for C++ */
extern "C" {          /* Linkage for C */
#endif
    extern Data* c_create_object(int);
    extern void c_access_object(Data*);
    extern void c_release_object(Data*);
    extern Data* call_create(int);
    extern int call_get(Data* data);
    extern void call_set(Data* data, int d);
    extern void call_release(Data*);
#endif /* Guard for C++ */
#endif /* __DATA_H */

Programming in Modern C++
```



C / C++ Mix Project: Data.cpp

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

```
// C++ code: Data.cpp
#include <iostream>
using namespace std;
#include "Data.h"

// Class Data implementation
Data::Data(int d): d_(d)
{
    cout << "Created " << d_ << endl;
}

Data::~Data()
{
    cout << "Released " << d_ << endl;
}

int Data::get()
{
    return d_;
}

void Data::set(int d)
{
    d_ = d;
}
```



C / C++ Mix Project: Data_Wrap.cpp

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

```
// C++ code: Data_Wrap.cpp
#include "Data.h"

/* C++ wrapper to create an object by new */
Data* call_create(int d)
{ return new Data(d); }

/* C++ wrapper to get state of an object by get */
int call_get(Data* data)
{ return data->get(); }

/* C++ wrapper to change state of an object by set */
void call_set(Data* data, int d)
{ return data->set(d); }

/* C++ wrapper to release an object by delete */
void call_release(Data* data)
{ delete data; }
```



C / C++ Mix Project: App.c

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

```
/* C code */: App.c
#include <stdio.h>
#include "Data.h"

Data* c_create_object(int d) { /* C function to create an object */
    return call_create(d);
}

void c_access_object(Data* data) { /* C function to access an object */
    printf("Get data %d\n", call_get(data));
    call_set(data, 7);
    printf("Set data %d\n", call_get(data));
}

void c_release_object(Data* data) { /* C function to release an object */
    call_release(data);
}
```



C / C++ Mix Project: main.cpp

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++
Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

```
// C++ code: main.cpp
#include <iostream>
using namespace std;

#include "Data.h"

Data d(10);

int main() {
    Data* p = c_create_object(5); /* C function to create an object */
    c_access_object(p); /* C function to access an object */
    c_release_object(p); /* C function to release an object */
}
```



C / C++ Mix Project: makefile

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

```
# Compiles .c by C and .cpp by C++. Links by C
CC=gcc
# Compiles .c and .cpp by C++. Links by C++
CPP=g++
CFLAGS=-I.
DEPS = Data.h

# Build .c by gcc (C Rules)
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

# Build .cpp by gcc (C++ Rules). May use $(CPP)$ for g++ also
%.o: %.cpp $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

# Link by g++ (C++ Linkage)
Data: main.o Data.o App.o Data_Wrap.o
    $(CPP) -o Data main.o Data.o App.o Data_Wrap.o

.PHONY: clean

clean:
    del *.o *.exe
```



C / C++ Mix Project: Execution

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

```
// Build by make
$ make
gcc -c -o main.o main.cpp -I.
gcc -c -o Data.o Data.cpp -I.
gcc -c -o App.o App.c -I.
gcc -c -o Data_Wrap.o Data_Wrap.cpp -I.
g++ -o Data main.o Data.o App.o Data_Wrap.o
// C++ Compile
// C++ Compile
// C Compile
// C++ Compile
// C++ Link
```

```
// Execute
$ Data.exe
Created 10
Created 5
Get data 5
Set data 7
Released 7
Released 10
```



C / C++ Mix Project: Call Trace

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

```
// Trace
START()
    Data::Data(int)

    main()
        c_create_object(int)
            call_create(int)
                new Data(int)
                    Data::Data(int)
c_access_object(Data*)
    printf(const char*, ...)
        call_get(Data*)
            Data::get()
call_set(Data*, int)
    Data::set(int)
printf(const char*, ...)
    call_get(Data*)
        Data::get()
c_release_object(Data*)
    call_release(Data*)
        delete(Data*)
            ~Data::Data()

    ~Data::Data()

// Special start-up function to initialize static objects in C++
// C++ constructor for class Data
// Start of main()
// C++ main() function
// C application function
// C++ wrapper
// C++ dynamic allocator
// C++ constructor for class Data
// C application function
// C library function
// C++ wrapper
// C++ member function for class Data
// C++ wrapper
// C++ member function for class Data
// C library function
// C++ wrapper
// C++ member function for class Data
// C application function
// C++ wrapper
// C++ dynamic de-allocator
// C++ destructor for class Data
// End of main()
// C++ destructor for class Data
```



Advanced Code Mix Scenarios

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

NPTEL

Advanced Code Mix Scenarios and Advisory



Advanced Code Mix Scenarios and Advisory

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++

Rules

Scenarios

C / C++ Mixed
Project

Data.h

Data.cpp

Data_Wrap.cpp

App.c

main.cpp

makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

- The common code mix scenarios as described:
 - Are *simple to code* and *easy to debug*
 - Covers *most situations* in several projects
 - Are *stable*, *portable*, and *recommended*
- Beyond this, however, several other scenarios may need resolution from time to time:
 - Using **exceptions** in C++ code
 - ▷ C++ exception mechanism and rules about *destroying objects that go out of scope* are likely to be *violated* by a C `longjmp`, with *unpredictable results*
 - ▷ **ADVISORY:** Do not to use `longjmp` in programs that contain C++ code
 - ▷ *Stack unwinding* while the control passes through a C function, is *not portable or stable*
 - ▷ **ADVISORY:** Avoid exception path going through any C function
 - Manipulating objects in a **polymorphic hierarchy** should be carefully handled as
 - ▷ *C does not support dynamic dispatch*
 - ▷ **ADVISORY:** Use appropriate C++ wrappers to avoid getting into C's *if-else type switch*
 - Directly **accessing data members** in classes from C. This can be really tricky because
 - ▷ *Layout of objects in a hierarchy is not portable*
 - ▷ **ADVISORY:** Access data members only through appropriately designed C++ wrappers
 - Several project / software specific scenarios



Tutorial Summary

Tutorial T06

Partha Pratim
Das

Tutorial Recap

Objectives &
Outline

Mixing C & C++
Rules

Scenarios

C / C++ Mixed
Project

Data.h
Data.cpp

Data_Wrap.cpp

App.c

main.cpp
makefile

Execution

Call Trace

Advanced Mix

Tutorial Summary

- We have learnt why is it often necessary to mix C and C++ codes in the same project
- We have explored the basic issues of mixing and learnt the ground rules
- In addition to the rules, we have four mechanisms to ease code mixing
 - Use `extern "C"` in C++ for all functions to be called from both C and C++
 - Guard `extern "C"` with `__cplusplus` guard for use with C
 - Provide `wrappers` for C++ data members, member functions, and overloaded functions for use with C
 - Incomplete `struct` type (with the same name as a C++ class) to allow pointers of C++ UDT objects in C
- We have also noted a few advanced mix scenarios and learnt the advisory of do's and don'ts



Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

Programming in Modern C++

Tutorial T07: How to design a UDT like built-in types?: Part 1: Fraction UDT

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 Objectives

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

NPTEL



Tutorial Outline

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

1 Data types

2 Fraction UDT

- Design

- Definition

- Operations

- Rules

- Class Design

- Version 1

- Design

- Implementation

- Test

- Version 2

- Design

- Implementation

- Pass Test

- Fail Test

- Cross Version

3 Tutorial Summary

Programming in Modern C++

Partha Pratim Das

T07.3



Data types

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

NPTEL

Data types



Notion of Data types

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Data types in C++ are used to specify the type of the data we use in our programs.
- They are classified under three categories:

- Built-In or Primitive Data types
- Derived Data types
- User-Defined Data type

- **Built-in data types:**

- Built-in data types are the most basic data types in C++
- They are predefined and can be used directly in a program
- **Examples:** `char`, `int`, `float` and `double`
- Apart from these, we also have `void` and `bool` data types

- **Derived Data types:**

- Data types that are derived from the built-in types
- **Examples:** arrays, functions, references and pointers

- **User Defined Type (UDT):**

- Those are declared & defined by the user using basic data types before using it
- **Examples:** structures, unions, enumerations and classes



User Defined Types

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Operator overloading helps us *build complete algebra* for UDT's much in the same line as is available for built-in types, called as, *Building data type*
 - **Complex type:** Add (+), Subtract (-), Multiply (*), Divide (/), Conjugate (!), Compare (==, !=, ...), etc.
 - **Fraction type:** Add (+), Subtract (-), Multiply (*), Divide (/), Reduce (unary *), Compare (==, !=, ...), etc.
 - **Matrix type:** Add (+), Subtract (-), Multiply (*), Divide (/), Invert (!), Compare (==, !=, ...), etc.
 - **Set type:** Union (+), Difference (-), Intersection (*), Subset (<, <=), Superset (>, >=), Compare (==, !=), etc.
 - **Direct IO:** read (>>) and write (<<) for all types



Fraction UDT

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

NPTEL

Fraction UDT



Design of Fraction UDT

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT
Design

Definition
Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- We intend to design a UDT **Fraction** which can behave like the build-in types like **int**
- The broad tasks involved include:
 - Make a clear statement of the concept of **Fraction**
 - Identify a representation for a **Fraction** object
 - Identify the properties and assertions applicable to all objects
 - Identify the operations for **Fraction** objects
 - ▷ Choose appropriate operators to overload for the operations
 - ▷ For example **operator+** to add two **Fraction** objects, or **operator<<** to stream a **Fraction** to **cout**
 - ▷ *Do not break the natural semantics for the operators*
- While it is possible to design and implement the UDT in one go (once you have acquired some expertise); it is better to go with iterative refinement. That is:
 - Make a design
 - Implement and Test
 - Refine and repeat



Notion of Fraction

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Intuitively fraction is a notation for numbers of the form $\frac{p}{q}$ where p and q are integers, like $\frac{2}{3}, \frac{4}{6}, \frac{3}{2}$ etc.
 - Fraction representation is *non-unique*: $\frac{2}{3} = \frac{4}{6} = \frac{8}{12} = \frac{-2}{-3}; \dots, -\frac{2}{3} = \frac{-2}{3} = \frac{2}{-3}$
- For our UDT design, we need *uniqueness of representation*. So let us restrict with the following rules for a fraction $\frac{p}{q}$:
 - q must be *positive*: $q > 0$
 - p and q must be *mutually prime*: $\gcd(p, q) = 1$
- Such fractions are known as *rational numbers* in mathematics
- Further a fraction $\frac{p}{q}$ is called *proper* if $|\frac{p}{q}| < 1$. It is *improper*, otherwise
 - An *improper fraction* can be written in *mixed fraction format* (assume $p > 0$) where we specify the maximum whole number in the fraction and the remaining proper fraction part:

$$\frac{p}{q} = (p \div q) \frac{p \% q}{q}$$

For example, $\frac{17}{3} = 5\frac{2}{3}$



Definition of Fraction

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

Definition

$\frac{p}{q}$ is a fraction where p and q are integers, $q > 0$, and p and q are mutually prime, that is,
 $\gcd(p, q) = 1$

That is, $p \in \mathcal{Z}$, $q \in \mathcal{N}$, $\gcd(p, q) = 1$, where \mathcal{Z} is the set of integers and \mathcal{N} is the set of natural numbers

p is called the numerator and q is called the denominator

Definition

Any fraction $\frac{p}{q}$ where $\gcd(p, q) > 1$, is irreduced and can be reduced to

$$\frac{p}{q} = \frac{p \div \gcd(p, q)}{q \div \gcd(p, q)}$$



Operations of Fraction

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Definition

Reduction:

$$\begin{aligned}\frac{p}{q} &= \frac{p/gcd(p, q)}{q/gcd(p, q)}, \text{ if } gcd(p, q) \neq 1 \\ &= \frac{-p}{-q}, \text{ if } q < 0 \\ &= \frac{0}{1}, \text{ if } p = 0 \\ &= \text{undefined, if } q = 0\end{aligned}$$

Addition: $(\frac{p}{q}) + (\frac{r}{s}) = \frac{p*(lcm(q,s)/q)+r*(lcm(q,s)/s)}{lcm(q,s)}$. Example 1: $\frac{5}{12} + \frac{7}{18} = \frac{5*3+7*2}{36} = \frac{29}{36}$

Subtraction: $(\frac{p}{q}) - (\frac{r}{s}) = (\frac{p}{q}) + (\frac{-r}{s})$. Example 2: $\frac{5}{12} - \frac{7}{18} = \frac{5*3+(-7)*2}{36} = \frac{1}{36}$

Multiplication: $(\frac{p}{q}) * (\frac{r}{s}) = \frac{p*r}{q*s}$. Example 3: $\frac{5}{12} * \frac{7}{18} = \frac{5*7}{12*18} = \frac{35}{216}$

Division: $(\frac{p}{q}) / (\frac{r}{s}) = \frac{p*s}{q*r}$. Example 4: $\frac{5}{12} / \frac{7}{18} = \frac{5*18}{7*12} = \frac{15}{14}$

Modulus: $(\frac{p}{q}) \% (\frac{r}{s}) = \frac{p}{q} - \lfloor (\frac{p}{q}) / (\frac{r}{s}) \rfloor * \frac{r}{s}$. Example 5: $\frac{5}{12} \% \frac{7}{18} = \frac{5}{12} - \lfloor \frac{15}{14} \rfloor * \frac{7}{18} = \frac{1}{36}$

where $lcm(q, s) = (q * s) / gcd(q, s)$

Partha Pratim Das

T07.11



Rules of Fraction

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

Fractions obey five rules of algebra as follows. For two fractions $\frac{p}{q}$ and $\frac{r}{s}$,

Definition

Rule of Invertendo: $\frac{p}{q} = \frac{r}{s} \Rightarrow \frac{q}{p} = \frac{s}{r}$. Use $! \frac{p}{q} = \frac{q}{p}$

Rule of Alternendo: $\frac{p}{q} = \frac{r}{s} \Rightarrow \frac{p}{r} = \frac{q}{s}$

Rule of Componendo: $\frac{p}{q} :: \frac{r}{s} \Rightarrow \frac{p+q}{q} :: \frac{r+s}{s}$. Use $++ \frac{p}{q} = \frac{p+q}{q} = \frac{p}{q} + 1$

Rule of Dividendo: $\frac{p}{q} :: \frac{r}{s} \Rightarrow \frac{p-q}{q} :: \frac{r-s}{s}$. Use $-- \frac{p}{q} = \frac{p-q}{q} = \frac{p}{q} - 1$

Rule of Componendo & Dividendo: $\frac{p}{q} :: \frac{r}{s} \Rightarrow \frac{p+q}{p-q} :: \frac{r+s}{r-s}$

We define three operations on fractions: Invertendo ([operator!](#)), Componendo ([operator++](#)), and Dividendo ([operator--](#)) to facilitate fraction algebra expressions



Design of Fraction Class

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition
Operations

Rules

Class Design

Version 1

Design

Implementation
Test

Version 2

Design

Implementation
Pass Test

Fail Test

Cross Version

Tutorial Summary

- From the definition, the representation of a **Fraction** can simply be:

```
class Fraction { // Implicit assertion for proper fraction: gcd(|n_|, d_) = 1
    int n_;           // numerator. n_ belongs to Z
    unsigned int d_; // denominator. d_ belongs to N
}
```

- Fraction** should support the following operation like **int**:

- Construction, Destruction and Copy Operations
- Unary Arithmetic Operations: Preserve (Sign), Negate, Componendo, and Dividendo
- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Mod
- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq
- IO Operations: Read and Write

- Fraction** should also support the following extended operation:

- Invert
- Convert to **double**

- Fraction** also need to support the following utilities for convenience:

- GCD and LCM
- Reduction (of irreduced fraction to reduced fraction)



Design of Fraction: Interface: Version 1

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Construction, Destruction, and Copy Operations

```
explicit Fraction(int = 1, int = 1); // Three overloads including a default constructor
~Fraction(); // No virtual destructor needed
Fraction(const Fraction&); // Copy constructor
Fraction& operator=(const Fraction&); // Copy assignment operator
```

- IO Operations: Read and Write

```
static void Write(const Fraction&); // Outstreams a fraction to cout in n/d form
static void Read(Fraction&); // Instreams n & d from cin to construct a fraction
```

- Unary Arithmetic Operations: Negate, Preserve (Sign), Componendo, and Dividendo

```
Fraction Negate() const; // Negate. p/q <- -p/q
Fraction Preserve() const; // Preserve. p/q <- p/q
Fraction& Componendo(); // Componendo. p/q <- p/q + 1
Fraction& Dividendo(); // Dividendo. p/q <- p/q - 1
```

- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
Fraction Add(const Fraction&) const; // Generates a result fraction,
Fraction Subtract(const Fraction&) const; // Does not change the current object
Fraction Multiply(const Fraction&) const;
Fraction Divide(const Fraction&) const;
Fraction Modulus(const Fraction&) const;
```



Design of Fraction: Interface: Version 1

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
bool Eq(const Fraction&) const;      // Generates a comparison result
bool NotEq(const Fraction&) const; // Does not change the current object
bool Less(const Fraction&) const;
bool LessEq(const Fraction&) const;
bool More(const Fraction&) const;
bool MoreEq(const Fraction&) const;
```

- Extended Operations: Invert and Convert to double

```
Fraction Invert() const; // Inverts a fraction. !(p/q) = q/p
double Double();        // Converts a fraction to a double
```

- Static constant fractions

```
static const Fraction UNITY; // Defines 1/1
static const Fraction ZERO; // Defines 0/1
```

- Support Functions: gcd, lcm and reduce: Should be private - not part of interface

```
static int gcd(int, int); // Finds the gcd for two +ve integers
static int lcm(int, int); // Finds the lcm for two +ve integers
Fraction& Reduce();       // Reduces a fraction
```



Implementation of Fraction: Version 1

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Construction, Destruction, and Copy Operations

```
explicit Fraction(int n = 1, int d = 1): // Three overloads
    n_(d < 0 ? -n : n), d_(d < 0 ? -d : d) // d_ is unsigned int. So no -ve value
{ Reduce(); } // Reduces the fraction
Fraction(const Fraction& f) : n_(f.n_), d_(f.d_) { } // Copy Constructor
~Fraction() { } // No virtual destructor needed
Fraction& operator=(const Fraction& f) { n_ = f.n_; d_ = f.d_; return *this; }
```

- IO Operations: Read and Write

```
static void Write(const Fraction& f) { cout << f.n_;
    if ((f.n_ != 0) && (f.d_ != 1)) cout << "/" << f.d_; // Suppress denominator
    // if n_ == 0 or d_ == 1
}
static void Read(Fraction& f) { cin >> f.n_ >> f.d_; f.Reduce(); }
```

- Unary Arithmetic Operations: Negate, Preserve (Sign), Componendo, and Dividendo

```
Fraction Negate() const { return Fraction(-n_, d_); }
Fraction Preserve() const { return *this; }
Fraction& Componendo() { return *this = Add(Fraction::UNITY); }
Fraction& Dividendo() { return *this = Subtract(Fraction::UNITY); }
```



Implementation of Fraction: Version 1

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition
Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
Fraction Add(const Fraction& f2) const {
    unsigned int d = Fraction::lcm(d_, f2.d_);
    int n = n_*(d / d_) + f2.n_*(d / f2.d_);
    return Fraction(n, d);
}
Fraction Subtract(const Fraction& f2) const { return Add(f2.Negate()); }
Fraction Multiply(const Fraction& f2) const {
    return Fraction(n_*f2.n_, d_*f2.d_);
}
Fraction Divide(const Fraction& f2) const { return Multiply(f2.Invert()); }
Fraction Modulus(const Fraction& f2) const {
    if (f2.n_ == 0) { throw "Divide by 0 is undefined\n"; }
    Fraction tf = Divide(f2);
    Fraction f = Subtract(Fraction(static_cast<int>(tf.n_ / tf.d_)).Multiply(f2));

    return f;
}
```



Implementation of Fraction: Version 1

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
bool Eq(const Fraction& f2) const { return ((n_ == f2.n_) && (d_ == f2.d_)); }
bool NotEq(const Fraction& f2) const { return !(Eq(f2)); }
bool Less(const Fraction& f2) const { return Subtract(f2).n_ < 0; }
bool LessEq(const Fraction& f2) const { return !More(f2); }
bool More(const Fraction& f2) const { return Subtract(f2).n_ > 0; }
bool MoreEq(const Fraction& f2) const { return !Less(f2); }
```

- Extended Operations: Invert and Convert to double

```
Fraction Invert() const { // Inverts a fraction. !(p/q) = q/p
    if (d_ == 0) throw "Divide by 0 is undefined\n";
    return Fraction(d_, n_);
}
double Double() const { // Converts to a double
    return static_cast<double>(n_) / static_cast<double>(d_);
}
```

- Static constant fractions

```
static const Fraction UNITY; // Defines 1/1
static const Fraction ZERO; // Defines 0/1
```



Implementation of Fraction: Version 1

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Support Functions: gcd, lcm and reduce: Should be **private** - not part of interface

```
static int gcd(int a, int b) { // Finds the gcd for two +ve integers
    while (a != b)
        if (a > b) a = a - b;
        else b = b - a;
    return a;
}
static int lcm(int a, int b) { // Finds the lcm for two +ve integers
    return (a / gcd(a, b))*b;
}
Fraction& Reduce() { // Reduces a fraction
    if (d_ == 0) { throw "Fraction with Denominator 0 is undefined"; }
    if (d_ < 0) { n_ = -n_;
        d_ = static_cast<unsigned int>(-static_cast<int>(d_));
        return *this;
    }
    if (n_ == 0) { d_ = 1; return *this; }
    unsigned int n = (n_ > 0) ? n_ : -n_, g = gcd(n, d_);
    n_ /= static_cast<int>(g); // as n_ is int and g is unsigned int the division may not work
    d_ /= g;
    return *this;
}
```



Testing Fraction: Version 1: Test Application

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition
Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

```
#include <iostream>
using namespace std;
#include "Fraction.h"

int main() {
    cout << "Construction, Copy Operations and Write Test" << endl; // Ctor, Copy & Write Test
    Fraction f1(5, 3); cout << "Fraction f1(5, 3) = "; Fraction::Write(f1); cout << endl;
    Fraction f2(7);    cout << "Fraction f2(7) = "; Fraction::Write(f2); cout << endl;
    Fraction f3;       cout << "Fraction f3 = "; Fraction::Write(f3); cout << endl;
    Fraction f4(f1);   cout << "Fraction f4(f1) = "; Fraction::Write(f4); cout << endl;
    Fraction f5(3, 6); cout << "Fraction f5(3, 6) = "; Fraction::Write(f5); cout << endl;
    Fraction f6(0, 4); cout << "Fraction f6(0, 4) = "; Fraction::Write(f6); cout << endl;
    cout << "Assignment: f2 = f1: f2 = "; Fraction::Write(f2 = f1); cout << endl << endl;

    cout << "Read Test" << endl; // Read Test
    cout << "Read f1 = "; Fraction::Read(f1); Fraction::Write(f1); cout << endl << endl;

    f1 = Fraction(2, 5); /* Using f1 for the following tests */ f2 = f1; // Copy to restore f1 later
    cout << "Unary Ops Test: Using f1 = ";
    Fraction::Write(f1); cout << " for all" << endl; // Unary Operations Test
    cout << "Negate: f1.Negate() = "; Fraction::Write(f1.Negate()); cout << endl;
    cout << "Preserve: f1.Preserve() = "; Fraction::Write(f1.Preserve()); cout << endl;
    cout << "Componendo: f1.Componendo() = "; Fraction::Write(f1.Componendo()); cout << endl; f1 = f2;
    cout << "Dividendo: f1.Dividendo() = "; Fraction::Write(f1.Dividendo()); cout << endl << endl;
```



Testing Fraction: Version 1: Test Application

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

```
f1 = Fraction(5, 12); f2 = Fraction(7, 18); // Using f1 and f2 for the following test
cout << "Binary Ops Test: Using f1 = "; // Binary Operations Test
Fraction::Write(f1); cout << ". f2 = "; Fraction::Write(f2); cout << " for all" << endl;
cout << "Binary Plus: f1.Add(f2) = "; Fraction::Write(f1.Add(f2)); cout << endl;
cout << "Binary Minus: f1.Subtract(f2) = "; Fraction::Write(f1.Subtract(f2)); cout << endl;
cout << "Binary Multiply: f1.Multiply(f2) = "; Fraction::Write(f1.Multiply(f2)); cout << endl;
cout << "Binary Divide: f1.Divide(f2) = "; Fraction::Write(f1.Divide(f2)); cout << endl;
cout << "Binary Residue: f1.Modulus(f2) = "; Fraction::Write(f1.Modulus(f2)); cout << endl << endl;

// Using f1 = Fraction(5, 12); f2 = Fraction(7, 18); for the following tests
cout << "Logical Ops Test: Using f1 = "; // Logical Operations Test
Fraction::Write(f1); cout << ". f2 = "; Fraction::Write(f2); cout << " for all" << endl;
cout << "Equal: " << ((f1.Eq(f2)) ? "true" : "false") << endl;
cout << "Not Equal: " << ((f1.NotEq(f2)) ? "true" : "false") << endl;
cout << "Less: " << ((f1.Less(f2)) ? "true" : "false") << endl;
cout << "Less Equal: " << ((f1.LessEq(f2)) ? "true" : "false") << endl;
cout << "Greater: " << ((f1.More(f2)) ? "true" : "false") << endl;
cout << "Greater Equal: " << ((f1.MoreEq(f2)) ? "true" : "false") << endl << endl;

// Using f1 = Fraction(5, 12); for the following tests
cout << "Extended Ops Test: Using f1 = "; // Extended Operations Test
Fraction::Write(f1); cout << " for all" << endl;
cout << "Invert: f1.Invert() = "; Fraction::Write(f1.Invert()); cout << endl;
cout << "Double: f1.Double() = "; cout << f1.Double() << endl << endl;
```



Testing Fraction: Version 1: Test Application

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

```
cout << "Static Constants Test" << endl; // Static Constants Test
cout << "UNITY = "; Fraction::Write(Fraction::UNITY); cout << endl;
cout << "ZERO = "; Fraction::Write(Fraction::ZERO); cout << endl << endl;
}

Construction, Copy Operations and Write Test
Fraction f1(5, 3) = 5/3
Fraction f2(7) = 7
Fraction f3 = 1
Fraction f4(f1) = 5/3
Fraction f5(3, 6) = 1/2
Fraction f6(0, 4) = 0
Assignment: f2 = f1: f2 = 5/3

Read Test
2 7
Read f1 = 2/7

Unary Ops Test: Using f1 = 2/5
Negate: f1.Negate() = -2/5
Preserve: f1.Preserve() = 2/5
Componendo: f1.Componendo() = 7/5
Dividendo: f1.Dividendo() = -3/5

All tests passed

Binary Ops Test: Using f1 = 5/12. f2 = 7/18 for all
Binary Plus: f1.Add(f2) = 29/36
Binary Minus: f1.Subtract(f2) = 1/36
Binary Multiply: f1.Multiply(f2) = 35/216
Binary Divide: f1.Divide(f2) = 15/14
Binary Residue: f1.Modulus(f2) = 1/36

Logical Ops Test: Using f1 = 5/12. f2 = 7/18 for all
Equal: false
Not Equal: true
Less: false
Less Equal: false
Greater: true
Greater Equal: true

Extended Ops Test: Using f1 = 5/12 for all
Invert: f1.Invert() = 12/5
Double: f1.Double() = 0.416667

Static Constants Test
UNITY = 1
ZERO = 0
```



Fraction: Version 1

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- So now we have one design and implementation for Fraction objects that can be manipulated by various operation member functions
- However, it still leaves a lot more to be desired. Consider, that we want to evaluate the following fraction expression:

$$f1 = \frac{2}{3}$$

$$f2 = \frac{8}{1}$$

$$f3 = \frac{5}{6}$$

$$f4 = (f1 + f2) / (f1 - f2) + !f3 - f2 * f3 = -\frac{1097}{165}$$

- Using Version 1:

```
void MixedText() { Fraction f1(2, 3), f2(8), f3(5, 6), f4;

    f4 = f1.Add(f2).Divide(f1.Subtract(f2)).Add(f3.Invert()).Subtract(f2.Multiply(f3));
    Fraction::Write(f4); cout << endl;
}
```

- *Horrendously complicated and error-prone, to say the least*

- To simplify, we map the member functions to various overloaded operators in Version 2



Design of Fraction: Interface: Version 2

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design
Definition
Operations
Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Construction, Destruction, and Copy Operations

```
explicit Fraction(int = 1, int = 1); // Three overloads including a default constructor
~Fraction(); // No virtual destructor needed
Fraction(const Fraction&); // Copy constructor
Fraction& operator=(const Fraction&); // Copy assignment operator
```

- IO Operations: Read and Write (`friend` function needed for `iostream` support)

```
friend ostream& operator<<(ostream&, const Fraction&); // Write()
friend istream& operator>>(istream&, Fraction&); // Read()
```

- Unary Arithmetic Operations: Preserve (Sign), Negate, Componendo, and Dividendo. Postfix operators are additions here

```
Fraction operator+() const; // Preserve()
Fraction operator-() const; // Negate()
Fraction& operator++(); // Pre-increment. Componendo(): p/q <- p/q + 1
Fraction& operator--(); // Pre-decrement. Dividendo(): p/q <- p/q - 1
Fraction operator++(int); // Post-increment.
                           // Lazy Componendo. p/q <- p/q + 1. Returns old p/q
Fraction operator--(int); // Post-decrement.
                           // Lazy Dividendo. p/q <- p/q - 1. Returns old p/q
```



Design of Fraction: Interface: Version 2

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
Fraction operator+(const Fraction&) const; // Add()
Fraction operator-(const Fraction&) const; // Subtract()
Fraction operator*(const Fraction&) const; // Multiply()
Fraction operator/(const Fraction&) const; // Divide()
Fraction operator%(const Fraction&) const; // Modulus()
```

Since the constructor of `Fraction` is `explicit`, an `int` cannot be implicitly converted to `Fraction`. So we do not expect an addition operation like `i + f` where `int i;` and `Fraction f`. Hence, member function operators are okay. Otherwise, we will need `friend` function operators:

```
friend Fraction operator+(const Fraction&, const Fraction&);
friend Fraction operator-(const Fraction&, const Fraction&);
friend Fraction operator*(const Fraction&, const Fraction&);
friend Fraction operator/(const Fraction&, const Fraction&);
friend Fraction operator%(const Fraction&, const Fraction&);
```



Design of Fraction: Interface: Version 2

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition
Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Advanced Assignment Operators. These are additions here:

```
Fraction& operator+=(const Fraction&);  
Fraction& operator-=(const Fraction&);  
Fraction& operator*=(const Fraction&);  
Fraction& operator/=(const Fraction&);  
Fraction& operator%=(const Fraction&);
```

- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
bool operator==(const Fraction&) const; // Eq()  
bool operator!=(const Fraction&) const; // NotEq()  
bool operator<(const Fraction&) const; // Less()  
bool operator<=(const Fraction&) const; // LessEq()  
bool operator>(const Fraction&) const; // More()  
bool operator>=(const Fraction&) const; // MoreEq()
```



Design of Fraction: Interface: Version 2

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Extended Operations: Invert and Convert to `double`

```
Fraction operator!() const; // Invert()  
operator double(); // Double()
```

- Static constant fractions

```
static const Fraction UNITY; // Defines 1/1  
static const Fraction ZERO; // Defines 0/1
```

- Support Functions: gcd, lcm and reduce: Should be `private` - not part of interface

```
static int gcd(int, int); // Finds the gcd for two +ve integers  
static int lcm(int, int); // Finds the lcm for two +ve integers  
Fraction& operator*(); // Reduce()
```

Since reduction is not on the interface, we may not overload an operator for it - it will be fine to use the earlier `Reduce()` function

- *Note that Bit-wise operators, Shift operators etc. are not overloaded in Fraction since there is no semantic interpretation for them*



Implementation of Fraction: Version 2

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT
Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Construction, Destruction, and Copy Operations

```
explicit Fraction(int n = 1, int d = 1) // Three overloads
    n_(d < 0 ? -n : n), d_(d < 0 ? -d : d) // d_ is unsigned int. So no -ve value
{ *(this); } // Reduces the fraction by operator*()
Fraction(const Fraction& f) : n_(f.n_), d_(f.d_) { } // Copy Constructor
~Fraction() { } // No virtual destructor needed
Fraction& operator=(const Fraction& f) { n_ = f.n_; d_ = f.d_; return *this; }
```

- IO Operations: Read and Write ([friend](#) function needed for [iostream](#) support)

```
friend ostream& operator<<(ostream& os, const Fraction& f) { os << f.n_;
    if ((f.n_ != 0) && (f.d_ != 1)) os << "/" << f.d_; // Suppress denominator
                                                // if n_ == 0 or d_ == 1
    return os;
}
friend istream& operator>>(istream& is, Fraction& f) { is >> f.n_ >> f.d_;
    *f; /* Reduces the fraction by operator*() */ return is;
}
```

- Unary Arithmetic Operations: Preserve, Negate, Componendo, Dividendo, & Postfix operators:

```
Fraction operator-() const { return Fraction(-n_, d_); }
Fraction operator+() const { return *this; }
Fraction& operator++() { return *this += Fraction::UNITY; }
Fraction& operator--() { return *this -= Fraction::UNITY; }
Fraction operator++(int) { Fraction f = *this; ++*this; return f; }
Fraction operator--(int) { Fraction f = *this; --*this; return f; }
```



Implementation of Fraction: Version 2

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition
Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulo:

```
Fraction operator+(const Fraction& f2) const {
    unsigned int d = lcm(d_, f2.d_);
    int n = n_*(d / d_) + f2.n_*(d / f2.d_);
    return Fraction(n, d);
}
Fraction operator-(const Fraction& f2) const { return *this + (-f2); }
Fraction operator*(const Fraction& f2) const { return Fraction(n_*f2.n_, d_*f2.d_); }
Fraction operator/(const Fraction& f2) const {
    if (f2.n_ == 0) { throw "Divide by 0 is undefined\n"; }
    return Fraction(n_*f2.d_, d_*f2.n_);
}
Fraction operator%(const Fraction& f2) const {
    if (f2.n_ == 0) { throw "Divide by 0 is undefined\n"; }
    Fraction tf = (*this) / f2;
    return (*this) - Fraction(tf.n_ % tf.d_, tf.d_)*f2;
    // return (*this) - Fraction(static_cast<int>(tf.n_ / tf.d_))*f2; // As in Ver 1
}
```



Implementation of Fraction: Version 2

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition
Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Advanced Assignment Operators. These are additions here:

```
Fraction& operator+=(const Fraction& f) { *this = *this + f; return *this; }
Fraction& operator-=(const Fraction& f) { *this = *this - f; return *this; }
Fraction& operator*=(const Fraction& f) { *this = *this * f; return *this; }
Fraction& operator/=(const Fraction& f) { *this = *this / f; return *this; }
Fraction& operator%=(const Fraction& f) { *this = *this % f; return *this; }
```

- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
bool operator==(const Fraction& f2) const { return ((n_ == f2.n_) && (d_ == f2.d_)); }
bool operator!=(const Fraction& f2) const { return !(*this == f2); }
bool operator<(const Fraction& f2) const { return (*this - f2).n_ < 0; }
bool operator<=(const Fraction& f2) const { return !(*this > f2); }
bool operator>(const Fraction& f2) const { return (*this - f2).n_ > 0; }
bool operator>=(const Fraction& f2) const { return !(*this < f2); }
```



Implementation of Fraction: Version 2

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Extended Operations: Invert and Convert to `double`

```
Fraction operator!() const { // Inverts a fraction. !(p/q) = q/p
    if (d_ == 0) { throw "Divide by 0 is undefined\n"; }
    return Fraction(d_, n_);
}
operator double() const { return static_cast<double>(n_) / static_cast<double>(d_); }
```

- Static constant fractions

```
static const Fraction UNITY; // Defines 1/1
static const Fraction ZERO; // Defines 0/1s
```

- Support Functions: gcd, lcm and reduce: Should be `private` - not part of interface

```
static int gcd(int a, int b);
static int lcm(int a, int b);
```

```
Fraction& operator*() { // Reduces a fraction
    if (d_ == 0) { throw "Fraction with Denominator 0 is undefined"; }
    if (d_ < 0) { n_ = -n_; d_ = static_cast<unsigned int>(-static_cast<int>(d_)); return *this; }
    if (n_ == 0) { d_ = 1; return *this; }
    unsigned int n = (n_ > 0) ? n_ : -n_, g = gcd(n, d_);
    n_ /= static_cast<int>(g); // as n_ is int and g is unsigned int the division may not work
    d_ /= g;
    return *this;
}
```



Testing Fraction: Version 2: Pass Test Application

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

```
#include <iostream>
using namespace std;
#include "Fraction.h"
int main() {
    cout << "Construction, Copy Operations and Write Test" << endl; // Ctor, Copy & and Write Test
    Fraction f1(5, 3); cout << "Fraction f1(5, 3) = " << f1 << endl;
    Fraction f2(7);    cout << "Fraction f2(7) = " << f2 << endl;
    Fraction f3;      cout << "Fraction f3 = " << f3 << endl;
    Fraction f4(f1); cout << "Fraction f4(f1) = " << f4 << endl;
    Fraction f5(3, 6); cout << "Fraction f5(3, 6) = " << f5 << endl;
    Fraction f6(0, 4); cout << "Fraction f6(0, 4) = " << f6 << endl;
    cout << "Assignment: f2 = f1: f2 = " << (f2 = f1) << endl << endl;

    cout << "Read Test" << endl; // Read Test
    cin >> f1; cout << "Read f1 = " << f1 << endl << endl;

    f1 = Fraction(2, 5); /* Using f1 for the following tests */ f2 = f1; // Copy to restore f1 later
    cout << "Unary Ops Test: Using f1 = " << f1 << " for all" << endl; // Unary Operations Test
    cout << "Negate: -f1 = " << -f1 << endl;
    cout << "Preserve: +f1 = " << +f1 << endl;
    cout << "Componendo: ++f1 = " << ++f1 << endl; f1 = f2;
    cout << "Dividendo: --f1 = " << --f1 << endl; f1 = f2;
    cout << "Lazy Componendo: f1++ = " << f1++ << " Lazy f1 = " << f1 << endl; f1 = f2;
    cout << "Lazy Dividendo: f1-- = " << f1-- << " Lazy f1 = " << f1 << endl << endl;
```



Testing Fraction: Version 2: Pass Test Application

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

```
f1 = Fraction(5, 12); f2 = Fraction(7, 18); // Using f1 and f2 for the following test

// Binary Operations Test
cout << "Binary Ops Test: Using f1 = " << f1 << ". f2 = " << f2 << " for all" << endl;
cout << "Binary Plus: f1 + f2 = " << (f1 + f2) << endl;
cout << "Binary Minus: f1 - f2 = " << (f1 - f2) << endl;
cout << "Binary Multiply: f1 * f2 = " << (f1 * f2) << endl;
cout << "Binary Divide: f1 / f2 = " << (f1 / f2) << endl;
cout << "Binary Residue: f1 % f2 = " << (f1 % f2) << endl << endl;

// Using f1 = Fraction(5, 12); f2 = Fraction(7, 18); for the following tests
f3 = f1; // Copy to restore f1 later
// Binary Assignment Operations Test
cout << "Binary Assignment Ops Test: Using f1 = " << f1 << ". f2 = " << f2 << " for all" << endl;
cout << "Plus Assign: f1 += f2: f1 = " << (f1 += f2) << endl; f1 = f3;
cout << "Minus Assign: f1 -= f2: f1 = " << (f1 -= f2) << endl; f1 = f3;
cout << "Multiply Assign: f1 *= f2: f1 = " << (f1 *= f2) << endl; f1 = f3;
cout << "Divide Assign: f1 /= f2: f1 = " << (f1 /= f2) << endl; f1 = f3;
cout << "Residue Assign: f1 %= f2: f1 = " << (f1 %= f2) << endl << endl; f1 = f3;
```



Testing Fraction: Version 2: Pass Test Application

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

```
// Using f1 = Fraction(5, 12); f2 = Fraction(7, 18); for the following tests
// Logical Operations Test
cout << "Logical Ops Test: Using f1 = " << f1 << ". f2 = " << f2 << " for all" << endl;
cout << "Equal: " << ((f1 == f2) ? "true" : "false") << endl;
cout << "Not Equal: " << ((f1 != f2) ? "true" : "false") << endl;
cout << "Less: " << ((f1 < f2) ? "true" : "false") << endl;
cout << "Less Equal: " << ((f1 <= f2) ? "true" : "false") << endl;
cout << "Greater: " << ((f1 > f2) ? "true" : "false") << endl;
cout << "Greater Equal: " << ((f1 >= f2) ? "true" : "false") << endl << endl;

// Extended Operations Test
// Using f1 = Fraction(5, 12); for the following tests
cout << "Extended Ops Test: Using f1 = " << f1 << " for all" << endl;
cout << "Invert: !f1 = " << !f1 << endl;
cout << "Double: (double)f1 = "; cout << static_cast<double>(f1) << endl << endl;

// Static Constants Test
cout << "Static Constants Test" << endl;
cout << "UNITY = " << Fraction::UNITY << endl;
cout << "ZERO = " << Fraction::ZERO << endl << endl;
}
```



Testing Fraction: Version 2: Pass Test Application

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition
Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

Construction, Copy Operations and Write Test
Fraction f1(5, 3) = 5/3
Fraction f2(7) = 7
Fraction f3 = 1
Fraction f4(f1) = 5/3
Fraction f5(3, 6) = 1/2
Fraction f6(0, 4) = 0
Assignment: f2 = f1: f2 = 5/3

Read Test

2 7

Read f1 = 2/7

Unary Ops Test: Using f1 = 2/5 for all

Negate: -f1 = -2/5

Preserve: +f1 = 2/5

Componendo: ++f1 = 7/5

Dividendo: --f1 = -3/5

Lazy Componendo: f1++ = 2/5 Lazy f1 = 7/5

Lazy Dividendo: f1-- = 2/5 Lazy f1 = -3/5

Binary Ops Test: Using f1 = 5/12. f2 = 7/18

Binary Plus: f1 + f2 = 29/36

Binary Minus: f1 - f2 = 1/36

All tests passed

Binary Multiply: f1 * f2 = 35/216
Binary Divide: f1 / f2 = 15/14
Binary Residue: f1 % f2 = 1/36

Binary Assignment Ops Test: Using f1 = 5/12. f2 = 7/18
Plus Assign: f1 += f2: f1 = 29/36
Minus Assign: f1 -= f2: f1 = 1/36
Multiply Assign: f1 *= f2: f1 = 35/216
Divide Assign: f1 /= f2: f1 = 15/14
Residue Assign: f1 %= f2: f1 = 1/36

Logical Ops Test: Using f1 = 5/12. f2 = 7/18 for all

Equal: false

Not Equal: true

Less: false

Less Equal: false

Greater: true

Greater Equal: true

Extended Ops Test: Using f1 = 5/12 for all

Invert: !f1 = 12/5

Double: (double)f1 = 0.416667

Static Constants Test

UNITY = 1

ZERO = 0 Partha Pratim Das



Testing Fraction: Version 2: Fail Test Application

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

```
int main() {
    try { cout << "Construct Fraction (1, 0): ";
        Fraction f1(1, 0); // Construct Fraction (1, 0): Fraction with Denominator 0 is undefined
    } catch (const char* s) { cout << s << endl; } cout << endl;
    Fraction f1;
    try { cout << "Read f1 = "; // Read f1 = 1 0
        cin >> f1; cout << f1 << endl; // Fraction with Denominator 0 is undefined
    } catch (const char* s) { cout << s << endl; } cout << endl;
    f1 = Fraction(5, 12); Fraction f2 = Fraction::ZERO, f3;
    try { cout << "Binary Divide: f3 = " << f1 << " / " << f2 << ": ";
        f3 = f1 / f2; cout << f3 << endl; // Binary Divide: f3 = 5/12 / 0: Divide by 0 is undefined
    } catch (const char* s) { cout << s << endl; }
    try { cout << "Binary Residue: f3 = " << f1 << " % " << f2 << ": ";
        f3 = f1 % f2; cout << f3 << endl; // Binary Residue: f3 = 5/12 % 0: Divide by 0 is undefined
    } catch (const char* s) { cout << s << endl; }
    try { cout << "Divide Assign: f1 = " << f1 << " /= " << f2 << ": ";
        f1 /= f2; cout << f1 << endl; // Divide Assign: f1 = 5/12 /= 0: Divide by 0 is undefined
    } catch (const char* s) { cout << s << endl; }
    try { cout << "Residue Assign: f1 = " << f1 << " %= " << f2 << ": ";
        f1 %= f2; cout << f1 << endl; // Residue Assign: f1 = 5/12 %= 0: Divide by 0 is undefined
    } catch (const char* s) { cout << s << endl; }
    try { cout << "Invert: f1 = " << " ! " << f2 << ": ";
        f1 = !f2; cout << f1 << endl; // Invert: f1 = ! 0: Fraction with Denominator 0 is undefined
    } catch (const char* s) { cout << s << endl; }
}
```



Cross Version Test

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- To assess Verion 2 against Version 1, again consider the following fraction expression:

$$f1 = \frac{2}{3}$$

$$f2 = \frac{8}{1}$$

$$f3 = \frac{5}{6}$$

$$f4 = (f1 + f2) / (f1 - f2) + !f3 - f2 * f3 = -\frac{1097}{165}$$

- Using Version 1: Very easy to get confused in the chain of calls and parentheses*

```
void MixedText() { Fraction f1(2, 3), f2(8), f3(5, 6), f4;
    f4 = f1.Add(f2).Divide(f1.Subtract(f2)).Add(f3.Invert()).Subtract(f2.Multiply(f3));
    Fraction::Write(f4); cout << endl;
}
```

- Using Version 2: Just as we write the algebra*

```
void MixedText() { Fraction f1(2, 3), f2(8), f3(5, 6), f4;
    f4 = (f1 + f2) / (f1 - f2) + !f3 - f2 * f3;
    cout << f4 << endl;
}
```



Tutorial Summary

Tutorial T07

Partha Pratim
Das

Objective &
Outline

Data types

Fraction UDT

Design

Definition

Operations

Rules

Class Design

Version 1

Design

Implementation

Test

Version 2

Design

Implementation

Pass Test

Fail Test

Cross Version

Tutorial Summary

- Analysed the difference between Built-in & UDT
- Discussed the meaning of Building a data type
- Understood the necessity of Building a data type
- Built a Fraction data type by iterative refinement



Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

Programming in Modern C++

Tutorial T08: How to design a UDT like built-in types?: Part 2: Int and Poly UDT

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 Recap

Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

- Analysed the difference between Built-in & UDT
- Discussed the meaning of Building a data type
- Understood the necessity of Building a data type
- Built a Fraction data type by iterative refinement



Tutorial Objectives

Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

- To build more UDTs: Int<N> and Poly<T>
- To test mix of UDTs

NPTEL



Tutorial Outline

Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

1 Tutorial Recap

2 Int<N> UDT

- Design
 - Operations
 - Class Design
- Implementation
- Test

3 Polynomial UDT

- Design
- Implementation
- Test

4 Practice UDTs

5 Tutorial Summary

NPTEL



Int<N> UDT

Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

NPTEL

Int<N> UDT



Understanding int

Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT
Design

Operations
Class Design

Implementation

Test

Polynomial UDT
Design

Implementation

Test

Practice UDTs

Tutorial Summary

- The datatype we are most familiar with is `int` which is a signed integer
 - Represented in a given number of bits, typically, 8, 16, 32, 64, or 128
 - Hence, `int` can represent numbers from `INT_MAX` to `INT_MIN`
 - For example, for 32 bits, $\text{INT_MAX} = 2^{31} - 1 = 2147483647$ and $\text{INT_MIN} = -2^{31} = -2147483648$
 - Beyond the `INT_MAX .. INT_MIN` range we get integer overflow and numbers wraparound

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    cout << INT_MAX   << endl; // 2147483647
    cout << INT_MIN   << endl; // -2147483648
    cout << INT_MAX+1 << endl; // -2147483648 integer overflow in expression of type 'int'
    cout << INT_MIN-1 << endl; // 2147483647 integer overflow in expression of type 'int'
    cout << -INT_MIN   << endl; // -2147483648 integer overflow in expression of type 'int'
}
```



Design of Int UDT

Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT
Design

Operations
Class Design

Implementation
Test

Polynomial UDT

Design
Implementation

Test

Practice UDTs

Tutorial Summary

- To understand `int` better, we intend to design a UDT `Int<N>` which can behave like `int` albeit for a size $N > 0$ bits that can be specified
- The range of values will be:
 - $\text{MinInt} = -2^{N-1} \dots \text{MaxInt} = 2^{N-1} - 1$
- The broad tasks involved include:
 - Make a clear statement of the concept of `Int`
 - Identify a representation for a `Int` object
 - Identify the properties and assertions applicable to all objects
 - Identify the operations for `Int` objects
 - ▷ Choose appropriate operators to overload for the operations
 - ▷ For example `operator+` to add two `Int` objects, or `operator<<` to stream a `Int` to `cout`
 - ▷ *Do not break the natural semantics for the operators*



Notion of Int

Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

- Intuitively `Int<N>` is a notation for whole numbers of the form of `N` bits signed integers
- $\text{MinInt} = -2^{N-1} \dots \text{MaxInt} = 2^{N-1} - 1$
- Numbers in `Int<N>` bits within a range of values `MinInt .. MaxInt`. Beyond this range the numbers wrap around:
 - $\text{MaxInt} + 1 = \text{MinInt}$
 - $\text{MinInt} - 1 = \text{MaxInt}$
- For example:
 - $N = 4 \Rightarrow \text{Range: } -8 \dots 7$
 - $\text{MinInt} = -2^3 = -8$ and $\text{MaxInt} = 2^3 - 1 = 7$
 - Except for `overflow`¹ as follows, all operations of `Int<N>` is same as `int`
 - ▷ $7 + 1 = -8$
 - ▷ $-8 - 1 = 7$
 - ▷ $-8 = -8$

¹Some authors distinguish between `overflow` (being more than `MaxInt`) and `underflow` (being less than `MinInt`). However, we prefer to use the term `overflow` in both cases because actually representation overflows the bits in both cases



Operations of Int<N>

Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

Definition

Limits:

$$\text{MaxInt} = 2^{N-1} - 1$$

$$\text{MinInt} = -2^{N-1}$$

$$\begin{aligned}-a &= a, \text{ if } a == \text{MinInt} \\ &= -a, \text{ otherwise}\end{aligned}$$

Negation:

Addition:

$$\begin{aligned}a + b &= a + b - 2^N, \text{ if } a + b > \text{MaxInt} \\ &= a + b + 2^N, \text{ if } a + b < \text{MinInt} \\ &= a + b, \text{ otherwise}\end{aligned}$$

Subtraction:

$$a - b = a + (-b)$$

Let $N = 4$

Example 1: $2 + 3 = 5$. $4 + 7 = -5$. $(-5) + 6 = 1$. $(-3) + (-2) = -5$. $(-6) + -7 = 3$

Example 2: $2 - 3 = -1$. $4 - 7 = -3$. $(-5) - 6 = 5$. $(-3) - (-2) = -1$. $(-6) - (-7) = 1$

Multiplication, Division, and Modulus: Left as exercise



Design of Int<N> Class

Tutorial T08

Partha Pratim
Das

Tutorial Recap

Objective &
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

- Clearly, the representation of a `Int<N>` needs to be a template with an `unsigned int` param `N`
- For the implementation, the `Int<N>` needs to use an underlying type `T` where basic arithmetic operations are available. So `T` is a type parameter for `Int<N>`. By default this can be `int`
- It is important to note that `N <= sizeof(T) * 8`. Otherwise, our basic operations may overflow
- Hence, the `Int<N>` class would look like:

```
template<typename T = int, unsigned int N = 4>
class Int_ { // an N-bits integer class with underlying type T
    T v_;      // actual value in underlying type T
    // ... Rest of the class
}
```

- Note that we name the type as `Int_` so that we can conveniently alias it in the user program as:
- ```
template<typename T, unsigned int N> class Int_;
typedef Int_<int, 4> Int; // T = int and N = 4
// Use as Int
```
- `Int<N>` should support the operation of `int`:
  - `Int<N>` should also support conversion the underlying type `T`
  - `Int<N>` may support the following constants for convenience of implementation:

```
static const Int_<T, N> MaxInt; // 2^(N-1)-1
static const Int_<T, N> MinInt; // -2^(N-1)
```



# Design of Int<N> Interface and Implementation

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

```
template<typename T = int, unsigned int N = 4>
class Int_ { public:
 static const Int_{T, N} MaxInt; // 2^(N-1)-1
 static const Int_{T, N} MinInt; // -2^(N-1)

 explicit Int_{T, N}(int v = 1) : v_(v) { // Two overloads of Constructor
 assert(v_ <= static_cast<int>(MaxInt)); // assert will fire if the value is out of limits
 assert(v_ >= static_cast<int>(MinInt));
 }
 Int_{T, N}(const Int_{T, N}& i) : v_(i.v_) { } // Copy Constructor
 ~Int_{T, N}() { } // No virtual destructor needed
 Int_{T, N}& operator=(const Int_{T, N}& i) { v_ = i.v_; return *this; } // Assignment
 // Streaming operators for IO
 friend ostream& operator<<(ostream& os, const Int_{T, N}& i) { os << i.v_; return os; }
 friend istream& operator>>(istream& is, Int_{T, N}& i) {
 T v; is >> v; i = Int_{T, N}(v); // We deliberately construct to test that v is within limits
 return is;
 }
 // Unary arithmetic operators
 Int_{T, N} operator-() const { return Int_{T, N}(v_ == MinInt_T? v_: -v_); }
 Int_{T, N} operator+() const { return *this; }
 Int_{T, N}& operator++() { *this = *this + Int_{T, N}(1); return *this; }
 Int_{T, N}& operator--() { *this = *this - Int_{T, N}(1); return *this; }
 Int_{T, N} operator++(int) { Int_{T, N} i = *this; ++*this; return i; }
 Int_{T, N} operator--(int) { Int_{T, N} i = *this; --*this; return i; }
```



# Design of Int<N> Interface and Implementation

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

```
// Binary arithmetic operators
friend Int_<T, N> operator+(const Int_<T, N>& i1, const Int_<T, N>& i2) {
 T v = i1.v_ + i2.v_; // add values in underlying type T
 if (v > MaxInt_T) // MaxInt_T is MaxInt in type T
 return Int_<T, N>(v - TwoPowerN_T); // wrap around if the value is more than MaxInt
 else if (v < MinInt_T) // MinInt_T is MinInt in type T
 return Int_<T, N>(v + TwoPowerN_T); // wrap around if the value is less than MinInt
 else
 return Int_<T, N>(v); // value within limits - no action
}
friend Int_<T, N> operator-(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1 + (-i2); }

// NOT IMPLEMENTED - left as exercises
friend Int_<T, N> operator*(const Int_<T, N>& i1, const Int_<T, N>& i2);
friend Int_<T, N> operator/(const Int_<T, N>& i1, const Int_<T, N>& i2);
friend Int_<T, N> operator%(const Int_<T, N>& i1, const Int_<T, N>& i2);

// Logical comparison operators
friend bool operator==(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ == i2.v_; }
friend bool operator!=(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ != i2.v_; }
friend bool operator<(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ < i2.v_; }
friend bool operator<=(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ <= i2.v_; }
friend bool operator>(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ > i2.v_; }
friend bool operator>=(const Int_<T, N>& i1, const Int_<T, N>& i2) { return i1.v_ >= i2.v_; }
```



# Design of Int<N> Interface and Implementation

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design  
Operations  
Class Design  
Implementation  
Test

Polynomial UDT

Design  
Implementation  
Test

Practice UDTs

Tutorial Summary

```
// Advanced assignment operators: NOT IMPLEMENTED - left as exercises
Int_<T, N>& operator+=(const Int_<T, N>& i);
Int_<T, N>& operator-=(const Int_<T, N>& i);
Int_<T, N>& operator*=(const Int_<T, N>& i);
Int_<T, N>& operator/=(const Int_<T, N>& i);
Int_<T, N>& operator%=(const Int_<T, N>& i);

operator T() const { return v_; } // conversion to underlying type T

private: // data members
 T v_; // Value in underlying type T
 static const T MaxInt_T; // MaxInt = 2^(N-1)-1 in underlying type T
 static const T MinInt_T; // MinInt = -2^(N-1) in underlying type T
 static const T TwoPowerN_T; // 2^N in underlying type T

public: static int Int_<T, N>::pow() { return Int_<T, N - 1>::pow() * 2; }
};

template<typename T> class Int_<T, 1> { public: static int Int_<T, 1>::pow() { return 1; } };

// Instantiations of static const members
const Int Int::MaxInt = Int(Int::pow() - 1);
const Int Int::MinInt = Int(-Int::pow());
const int Int::MaxInt_T = static_cast<int>(Int::MaxInt); // 2^(N-1)-1
const int Int::MinInt_T = static_cast<int>(Int::MinInt); // -2^(N-1)
const int Int::TwoPowerN_T = (Int::MaxInt_T+1) << 1; // 2^N
```



# Testing Int<N>

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

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

template<typename T, unsigned int N> class Int_;
typedef Int_<int, 4> Int; // N = 4

#include "Int.h"

void main() {
 cout << "Int::MaxInt = " << Int::MaxInt << endl; // Int::MaxInt = 7
 cout << "Int::MinInt = " << Int::MinInt << endl; // Int::MinInt = -8

 // Constructor, Copy Operations and Write Test
 Int f1(5); cout << "Int f1(5) = " << f1 << endl; // Int f1(5) = 5
 Int f2(0); cout << "Int f2(0) = " << f2 << endl; // Int f2(0) = 0
 Int f3; cout << "Int f3 = " << f3 << endl; // Int f3 = 1
 Int f4(f1); cout << "Int f4(f1) = " << f4 << endl; // Int f4(f1) = 5
 cout << "Assignment: f2 = f1: f2 = " << (f2 = f1) << endl; // Assignment: f2 = f1: f2 = 5

 // Read Test
 cin >> f1; // 3
 cout << "Read f1 = " << f1 << endl; // Read f1 = 3
```



# Testing Int<N>

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

```
// Unary Operations Test
cout << "-Int(2) = " << -Int(2) << endl; // -Int(2) = -2
cout << "-Int(-2) = " << -Int(-2) << endl; // -Int(-2) = 2
cout << "-Int(-8) = " << -Int(-8) << endl; // -Int(-8) = -8
cout << "-Int(7) = " << -Int(7) << endl; // -Int(7) = -7
cout << "++Int(2) = " << ++Int(2) << endl; // ++Int(2) = 3
cout << "++Int(7) = " << ++Int(7) << endl; // ++Int(7) = -8
cout << "++Int(-8) = " << ++Int(-8) << endl; // ++Int(-8) = -7
cout << "--Int(0) = " << --Int(0) << endl; // --Int(0) = -1
cout << "--Int(-7) = " << --Int(-7) << endl; // --Int(-7) = -8
cout << "--Int(-8) = " << --Int(-8) << endl; // --Int(-8) = 7

// Binary Operations Test
cout << "Int(2) + Int(3) = " << (Int(2) + Int(3)) << endl; // Int(2) + Int(3) = 5
cout << "Int(4) + Int(7) = " << (Int(4) + Int(7)) << endl; // Int(4) + Int(7) = -5
cout << "Int(-5) + Int(6) = " << (Int(-5) + Int(6)) << endl; // Int(-5) + Int(6) = 1
cout << "Int(-3) + Int(-2) = " << (Int(-3) + Int(-2)) << endl; // Int(-3) + Int(-2) = -5
cout << "Int(-6) + Int(-7) = " << (Int(-6) + Int(-7)) << endl; // Int(-6) + Int(-7) = 3
```



# Testing Int<N>

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

```
// Binary Operations Test
cout << "Int(2) - Int(3) = " << (Int(2) - Int(3)) << endl; // Int(2) - Int(3) = -1
cout << "Int(4) - Int(7) = " << (Int(4) - Int(7)) << endl; // Int(4) - Int(7) = -3
cout << "Int(-5) - Int(6) = " << (Int(-5) - Int(6)) << endl; // Int(-5) - Int(6) = 5
cout << "Int(-3) - Int(-2) = " << (Int(-3) - Int(-2)) << endl; // Int(-3) - Int(-2) = -1
cout << "Int(-6) - Int(-7) = " << (Int(-6) - Int(-7)) << endl; // Int(-6) - Int(-7) = 1

// Logical Operations Test
cout << "Int(-6) == Int(-6): " << ((Int(-6) == Int(-6)) ? "T" : "F") << endl; // Int(-6) == Int(-6): T
cout << "Int(4) != Int(3): " << ((Int(4) != Int(3)) ? "T" : "F") << endl; // Int(4) != Int(3): F
cout << "Int(-7) < Int(2): " << ((Int(-7) < Int(2)) ? "T" : "F") << endl; // Int(-7) < Int(2): F
cout << "Int(-7) <= Int(2): " << ((Int(-7) <= Int(2)) ? "T" : "F") << endl; // Int(-7) <= Int(2): F
cout << "Int(-5) > Int(-6): " << ((Int(-5) > Int(-6)) ? "T" : "F") << endl; // Int(-5) > Int(-6): F
cout << "Int(4) >= Int(4): " << ((Int(4) >= Int(4)) ? "T" : "F") << endl; // Int(4) >= Int(4) = T
}
```



# Polynomial UDT

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

NPTEL

## Polynomial UDT



# Design of Polynomial UDT

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

- Polynomials  $A(x)$  of  $x$  having degree  $\text{degree}(A) = n$  and  $n + 1$  coefficients  $a_0, a_1, a_2, \dots, a_n$ :

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n = \sum_{i=0}^n a_i x^i$$

- The representation of a polynomial UDT **Poly** would need
  - a vector to keep the coefficients, and
  - a simple member to the degree (for null polynomials without coefficients)
- The types of coefficient and variable should be appropriate so that they can be multiplied and added. For simplicity, let us assume that they have the same type:

```
template<typename T = int> // Type of Coefficients and value
class Poly { // a polynomial of type T
 vector<T> coeff_; // coefficients
 size_t deg_; // deg_ = coeff_.size()-1
 // ... Rest of the class
}
```



# Design of Polynomial UDT

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

- A polynomial  $A(x)$  of degrees  $n$  may be negated to generate polynomial  $R(x)$  of degrees  $n$  by flipping the sign of every coefficient. That is:

$$R(x) = -A(x) = - \sum_{i=0}^n a_i x^i = \sum_{i=0}^n (-a_i) x^i$$

Hence,

$$r_i = -a_i, \quad 0 \leq i \leq n$$

- Two polynomials  $A(x)$  and  $B(x)$  of degrees  $n$  and  $m$  respectively may be added to generate polynomial  $R(x)$  of degree  $\max(n, m)$  by pairwise adding the coefficients of the same power. That is, for  $n \geq m$

$$R(x) = A(x) + B(x) = \sum_{i=0}^n a_i x^i + \sum_{i=0}^m b_i x^i = \sum_{i=0}^m (a_i + b_i) x^i + \sum_{i=m+1}^n a_i x^i$$

Hence,

$$\begin{aligned} r_i &= a_i + b_i, \quad 0 \leq i \leq m \\ &= a_i, \quad m+1 \leq i \leq n \end{aligned}$$

Note:  $A(x) - B(x) = A(x) + (-B(x))$



# Design of Poly<T> Interface and Implementation

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

```
template<typename T = int> // Type of Coefficients and Value
class Poly { public:
 Poly(vector<T>& c = vector<T>(1)) : coeff_(c), deg_(c.size() - 1) { }
 Poly(size_t n) : coeff_(vector<T>(n+1)), deg_(n) { } // null polynomial
 Poly(const Poly& p) : coeff_(p.coeff_), deg_(p.deg_) { }
 ~Poly() { } // No virtual destructor needed
 Poly& operator=(const Poly& p) { deg_ = p.deg_; coeff_ = p.coeff_; return *this; }

 Poly operator+() const { return *this; }
 Poly operator-() const { Poly r(deg_);
 for (unsigned int i = 0; i <= deg_; i++) r.coeff_[i] = -coeff_[i];
 return r;
 }
 Poly operator+(const Poly& p) const { Poly r(max(p.deg_, deg_)); // result
 vector<T> v;
 if (deg_ > p.deg_) { v = p.coeff_; r.coeff_ = coeff_; } // copy the longer (shorter) vector
 else { v = coeff_; r.coeff_ = p.coeff_; } // of coefficients to r.coeff_(v)
 for (unsigned int i = 0; i <= min(p.deg_, deg_); i++) { // add the common coefficients
 r.coeff_[i] = t.coeff_[i] + v[i];
 }
 return r;
 }
 Poly operator-(const Poly&p) const { return *this + (-p); }
```



# Design of Poly<T> Interface and Implementation

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

```
Poly& operator+=(const Poly& p) { *this = *this + p; return *this; }
Poly& operator-=(const Poly&p) { *this = *this - p; return *this; }

friend ostream& operator<<(ostream& os, const Poly& p) { int j;
 for (j = p.deg_; j >= 0; --j)
 { if (static_cast<T>(0) != p.coeff_[j]) break; } // first non-zero coeff.
 if (0 > j)
 os << 0;
 else
 if (0 == j) os << p.coeff_[j];
 else os << p.coeff_[j] << "x^" << j;
 for (int i = j-1; i >= 0; --i) {
 if (static_cast<T>(0) != p.coeff_[i]) {
 if (0 != i)
 if (static_cast<T>(1) != p.coeff_[i])
 os << " + " << p.coeff_[i] << "x^" << i;
 else
 os << " + " << "x^" << i;
 else
 os << " + " << p.coeff_[i];
 }
 }
 os << ".";
 return os;
}
```



# Design of Poly<T> Interface and Implementation

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

```
friend istream& operator >>(istream& is, Poly& p) {
 cout << "Enter degree of the polynomial ";
 is >> p.deg_;
 p.coeff_.resize(p.deg_ + 1);
 cout << "Enter all the coefficients like a0+a1*x+a2*x^2+....an*x^n" << endl;
 for (unsigned int i = 0; i <= p.deg_; i++)
 is >> p.coeff_[i];
 return is;
}

// Evaluates the polynomial - use Horner's Rule
T operator()(const T& x) {
 T val = 0;
 for (int i = deg_; i >= 0; i--)
 val = val * x + coeff_[i];
 return val;
}

private:
 vector<T> coeff_;
 size_t deg_;
 template<typename T> inline static T max(T a, T b) { return a > b ? a : b; }
 template<typename T> inline static T min(T a, T b) { return a < b ? a : b; }
};
```



# Testing Poly<T>

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

```
#include <iostream>
using namespace std;
#include "fraction.h"
#include "polynomial.h"

void main() { vector<int> v = { 1, 2, 1 };
 Poly<int> p(v); cout << "p(x): " << p << " p(2) = " << p(2); // p(x): 1x^2 + 2x^1 + 1. p(2) = 9
 Poly<int> q(p); cout << "q(p): " << q << " q(2) = " << q(2); // q(p): 1x^2 + 2x^1 + 1. q(2) = 9
 Poly<int> s(vector<int>({ 0, 0, 1, 2, 1, 0, 2, 7, 0 }));
 cout << "s(x): " << s << " s(1) = " << s(1); // s(x): 7x^7 + 2x^6 + x^4 + 2x^3 + x^2. s(1) = 13
 Poly<int> r; cout << "r: " << r << " r(2) = " << r(2); // r: 1. r(2) = 1

 cin >> r; // 2 1 2 1
 cout << "r: " << r << " r(2) = " << r(2); // r: 1x^2 + 2x^1 + 1. r(2) = 9

 Poly<int> t(2); cout << "t(x): " << t << " t(2) = " << t(2); // t(x): 0. t(2) = 0

 r = p; cout << "r = p: " << r << " r(2) = " << r(2); // r = p: 1x^2 + 2x^1 + 1. r(2) = 9
 r = -p; cout << "r = -p: " << r << " r(2) = " << r(2); // r = -p: -1x^2 + -2x^1 + -1. r(2) = -9

 p = vector<int>({ 1, 5, 6 });
 cout << "p(x): " << p << " p(2) = " << p(2); // p(x): 6x^2 + 5x^1 + 1. p(2) = 35

 q = vector<int>({ 1, -2, 1 });
 cout << "q(x): " << q << " q(2) = " << q(2); // q(x): 1x^2 + -2x^1 + 1. q(2) = 1
```



# Testing Poly<T>

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

```
r = p + q; cout << "r = p + q: " << r << " r(2) = " << r(2); // r = p + q: 7x^2 + 3x^1 + 2. r(2) = 36
r = p - q; cout << "r = p - q: " << r << " r(2) = " << r(2); // r = p - q: 5x^2 + 7x^1. r(2) = 34
r = p; p += q; cout << "p += : " << p << " p(2) = " << p(2); // p += : 7x^2 + 3x^1 + 2. p(2) = 36
p = r; p -= q; cout << "p -= : " << p << " p(2) = " << p(2); // p -= : 5x^2 + 7x^1. p(2) = 34

vector<Fraction> vf = { Fraction(1, 2), Fraction(-3, 5), Fraction(2, 3) };
Poly<Fraction> pf1(vf); cout << "pf1(x): " << pf1 << " pf1(2) = " << pf1(2) << endl;
// pf1(x): 2/3x^2 + -3/5x^1 + 1/2. pf1(2) = 59/30

Poly<Fraction> pf2; cout << "pf2(x): " << pf2 << " pf2(2) = " << pf2(2) << endl;
// pf2(x): 1. pf2(2) = 1

cin >> pf2; // 1 2 3 1 2
cout << "pf2: " << pf2 << " pf2(2) = " << pf2(2) << endl;
// pf2(x): 1/2x^1 + 2/3. pf2(2) = 5/3

Poly<Fraction> pf3 = pf1 + pf2;
cout << "pf3(x): " << pf3 << " pf3(2) = " << pf3(2) << endl;
// pf3(x): 2/3x^2 + -1/10x^1 + 7/6. pf3(2) = 109/30

Poly<Fraction> pf4 = pf1 - pf2;
cout << "pf4(x): " << pf4 << " pf4(2) = " << pf4(2) << endl;
// pf4(x): 2/3x^2 + -11/10x^1 + -1/6. pf4(2) = 3/10
}
```



# Practice UDTs

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

NPTEL

## Practice UDTs



# Practice UDTs

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

- **Fraction**

- Change binary arithmetic and comparison operators to **friend** functions from methods
- Parameterize **Fraction** with type **T = int**
- Provide mixed mode support

- **Int<N>**

- Implement **operator\***()
- Implement **operator/()**
- Implement **operator%()**

- **Poly<T>**

- Test **Poly<double>**
- Use member function template for **operator()()** with another type parameter **U** for **x**
- Analyze the compatibility issues between types **T** and **U**

- Mixed UDTs

- Test **Fraction<Int<N> >**
- Test **Poly<Int<N> >**
- Test **Poly<Fraction<Int<N> > >**



# Tutorial Summary

Tutorial T08

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Int<N> UDT

Design

Operations

Class Design

Implementation

Test

Polynomial UDT

Design

Implementation

Test

Practice UDTs

Tutorial Summary

- Presented the design, implementation and test for Int<N> and Poly<T> types
- Showed how Poly<int> as well as Poly<Fraction> works
- Outlined several practice UDTs for homework

NPTEL



Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators  
Template  
Mixed Format

Int<N> UDT  
Wraparound  
Binary Ops

Mixed UDT Apps  
Fraction <int>  
Fraction <Int<4> >  
Poly<Int<4> >  
Poly <Fraction <Int<N> > >

Caveat

Tutorial Summary

# Programming in Modern C++

Tutorial T09: How to design a UDT like built-in types?: Part 3: Updates and Mixes of UDTs

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 Recap

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

- Presented the design, implementation and test for Int<N> and Poly<T> types
- Showed how Poly<int> as well as Poly<Fraction> works
- Outlined several practice UDTs for homework

NPTEL



# Tutorial Objectives

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >

Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

NPTEL



# Tutorial Outline

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template  
Mixed Format

Int<N> UDT

Wraparound  
Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4>>

Poly<Int<4>>

Poly <Fraction

<Int<N>>>

Caveat

Tutorial Summary

## 1 Tutorial Recap

## 2 Update UDTs

## 3 Fraction UDT

- friend Operators
- Template
- Mixed Format

## 4 Int<N> UDT

- Wraparound
- Binary Ops

## 5 Mixed UDT Apps

- Fraction <int>
- Fraction <Int<4>>
- Poly<Int<4>>
- Poly <Fraction <Int<N>>>

## 6 Caveat

## 7 Tutorial Summary



# Update UDTs

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

NPTEL

## Update UDTs



# Update UDTs

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template  
Mixed Format

Int<N> UDT

Wraparound  
Binary Ops

Mixed UDT Apps

Fraction <int>  
Fraction  
<Int<4> >  
Poly<Int<4> >  
Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

- **Fraction**

- Change binary arithmetic and comparison operators to **friend** functions from non-static member functions
- Parameterize **Fraction** with type **T = int**
- Provide mixed format support

- **Int<N>**

- In the constructor of **Int<N>**, allow out-of-range values to wrap around instead of **assert**
- Implement **operator\*()**, **operator/()**, and **operator%()**

- Mixed UDT Apps

- Test **Fraction<int>**
- Test **Fraction<Int<4> >**
- Test **Poly<Int<4> >**
- Test **Poly<Fraction<Int<4> > >**



# Fraction UDT: Update

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

**Fraction UDT**

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

NPTEL

## Fraction UDT: Update



# Fraction UDT: Update: Agenda

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

- We have the following update agenda for `Fraction`
  - Change binary arithmetic and comparison operators to `friend` functions from non-static member functions
  - Parameterize `Fraction` with type `T = int`
  - Provide mixed format support



# Fraction: friend Operators

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

- To facilitate the power of **friend** operators, we make the constructor non-explicit

```
/* explicit */ Fraction(int n = 1, int d = 1): // Three overloads
 n_(d < 0 ? -n : n), d_(d < 0 ? -d : d) // d_ cannot be -ve
{ *(this); } // Reduces the fraction by operator*()
```

- Binary Arithmetic Operations: Add, Subtract, Multiply, Divide, and Modulus

```
friend Fraction operator+(const Fraction&, const Fraction&); // Add()
friend Fraction operator-(const Fraction&, const Fraction&); // Subtract()
friend Fraction operator*(const Fraction&, const Fraction&); // Multiply()
friend Fraction operator/(const Fraction&, const Fraction&); // Divide()
friend Fraction operator%(const Fraction&, const Fraction&); // Residue()
```

- Binary Relational Operations: Less, LessEq, More, MoreEq, Eq, NotEq

```
friend bool operator==(const Fraction& f1, const Fraction& f2); // Eq()
friend bool operator!=(const Fraction& f1, const Fraction& f2); // NotEq()
friend bool operator<(const Fraction& f1, const Fraction& f2); // Less()
friend bool operator<=(const Fraction& f1, const Fraction& f2); // LessEq()
friend bool operator>(const Fraction& f1, const Fraction& f2); // More()
friend bool operator>=(const Fraction& f1, const Fraction& f2); // MoreEq()
```



# Fraction: Template

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction <Int<4> >

Poly<Int<4> >

Poly <Fraction <Int<N> >>

Caveat

Tutorial Summary

- To provide an underlying type for `Fraction`, we introduce type variable `T` with `int` as default
- `T` could be any integral type like `int`, `short`, `char`, `long`, or `Int<N>` etc.
- We also change the name of the type from `Fraction` to `Fraction_` not to clutter the user name space

```
template<typename T = int>
class Fraction_ { public:

 // Change Fraction to Fraction_

 // ...
private:
 T n_; // The Numerator. Earlier:int
 T d_; // The Denominator. Earlier: unsigned int
 // ...
};
```

- In the application, add:

```
typedef Fraction_<int> Fraction; // Fraction is used in the application
```



# Fraction: Template

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >

Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

```
template<typename T = int> class Fraction_ { public: // Parameterized Fraction_ with T = int
 Fraction_(T n = 1, T d = 1); Fraction_(const Fraction_& f); // Ctor, C-Ctor, C=, Dtor
 ~Fraction_(); Fraction_& operator=(const Fraction_& f);
 friend ostream& operator<<(ostream& os, const Fraction_& f); // Streaming ops
 friend istream& operator>>(istream& is, Fraction_& f);
 Fraction_ operator-() const; Fraction_ operator+() const; // Unary arithmetic ops
 Fraction_& operator++(); Fraction_ operator++(int);
 Fraction_& operator--(); Fraction_ operator--(int);
 friend Fraction_ operator+(const Fraction_& f1, const Fraction_& f2); // Binary arithmetic ops
 friend Fraction_ operator-(const Fraction_& f1, const Fraction_& f2);
 friend Fraction_ operator*(const Fraction_& f1, const Fraction_& f2);
 friend Fraction_ operator/(const Fraction_& f1, const Fraction_& f2);
 friend Fraction_ operator%(const Fraction_& f1, const Fraction_& f2);
 friend bool operator==(const Fraction_& f1, const Fraction_& f2); // Comparison ops
 friend bool operator!=(const Fraction_& f1, const Fraction_& f2);
 friend bool operator<(const Fraction_& f1, const Fraction_& f2);
 friend bool operator<=(const Fraction_& f1, const Fraction_& f2);
 friend bool operator>(const Fraction_& f1, const Fraction_& f2);
 friend bool operator>=(const Fraction_& f1, const Fraction_& f2);
 Fraction_& operator+=(const Fraction_& f); // Advanced assignment ops
 Fraction_& operator-=(const Fraction_& f); Fraction_& operator**=(const Fraction_& f);
 Fraction_& operator/=(const Fraction_& f); Fraction_& operator%=(const Fraction_& f);
 Fraction_ operator!() const; operator double() const; // Special ops
private: static T gcd(T a, T b); static T lcm(T a, T b); Fraction_& operator*(); // Support functions
};
```



# Fraction: Mixed Format Support

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >  
Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

- Irrespective of whether a fraction is in *simple format* like  $\frac{n}{d}$  ( $\frac{2}{3}$  or  $\frac{17}{5}$ ) or in *mixed format* like  $w\frac{n}{d}$  ( $\frac{2}{3}$  or  $3\frac{2}{5}$ ), its *internal representation is always simple*  $w\frac{n}{d} \equiv \frac{w*d+n}{d}$
- Hence, mixed format support is limited to:

- Fraction construction

```
explicit Fraction_(T w, T n, T d) : // Mixed format fraction constructor
 n_(d < 0 ? w * -d - n : w * d + n), d_(d < 0 ? -d : d) // d must be non-negative
 { *(this); } // Reduces the fraction
```

- Fraction output operator

```
friend ostream& operator<<(ostream& os, const Fraction_& f);
```

- Fraction input operator

```
friend istream& operator>>(istream& is, Fraction_& f);
```

- While the constructor can be distinguished by the distinct signature, the streaming operators have the same signature for *simple* as well as *mixed format*. Hence, we need a way to tell these operators about the format



# Fraction: Mixed Format Support in Streaming Operators

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

- To design for the mixed format i/o, we recall the support for writing integers in multiple bases using `<iomanip>` component in standard library

```
#include <iostream>
#include <iomanip>
int main() { int i = 76;
 std::cout << std::oct << i << std::endl; // Set octal format. Prints 114
 std::cout << std::hex << i << std::endl; // Set hexadecimal format. Prints 4c
 std::cout << std::dec << i << std::endl; // Set decimal format. Prints 76
}
```

- Using `<iomanip>`, the format flag is set in `ostream (cout)`. We cannot do that as `ostream` (or `istream`) cannot be changed. So, we need to keep the format option in the `Fraction_` class
- We add a `static bool bMixedFormat_` (`true` for mixed format, `false` for simple format)
- In the streaming operators, we can check this flag and adopt the appropriate formatting
- But how do we set / reset this flag? Using `SetFormat(bool)` spoils the built-in type-like syntax

Easy

```
Fraction f(17,5);
Fraction::SetFormat(false);
cout << f; // 17/5
Fraction::SetFormat(true);
cout << f; // 3+2/5
```

Desired

```
Fraction f(17,5);
cout << Fraction::simple;
cout << f; // 17/5
cout << Fraction::mixed;
cout << f; // 3+2/5
```



# Fraction: Mixed Format Support in Streaming Operators

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

- For `cout << Fraction::simple (cout << Fraction::mixed)`, we observe the following:
  - `Fraction::simple (Fraction::mixed)` needs to have an appropriate type, say `Format`, different from `Fraction_`, yet encapsulated by `Fraction_`. So we choose nested `class Format` in `Fraction_`

```
class Format { bool bFormat_; public: Format(bool b): bFormat_(b) { } /* ... */ }; // Wraps bool
```
  - `Fraction::simple` and `Fraction::mixed` must be constants in `Fraction_`

```
static const Format mixed; // bMixedFormat_ = true
static const Format simple; // bMixedFormat_ = false
```
  - Output streaming `Fraction::simple (Fraction::mixed)` in `Fraction::Format` should print nothing and set `Fraction_::bMixedFormat_` appropriately

```
friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
 // sets / resets mixed format flag
 bMixedFormat_ = m.bFormat_; // error: operator<< is friend of Format, not of Fraction_
 return os;
}
```
  - So we use a wrapper in `Format`

```
class Format { // ...
 void SetMixedFormat(bool b) const { bMixedFormat_ = b; } // access private member of Fraction_
 friend ostream& operator<<(ostream& os, const Format& m) { // writes nothing
 m.SetMixedFormat(m.bFormat_); // sets / resets mixed format flag
 return os;
 }
};
```



# Fraction: Mixed Format Support in Streaming Operators

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

- We use `Fraction_::bMixedFormat_` to decide the format in the streaming operators:

```
friend ostream& operator<<(ostream& os, const Fraction_& f) {
 T w = 0, n = f.n_, d = f.d_;
 if (f.bMixedFormat_) { // Mixed format support
 w = n / d; // Whole part = 3 in 17/5
 n %= d; // Fraction part = 2/5 in 17/5
 if (n < 0) { --w; n += d; } // Negative: -17/5 = -4+3/5 = (-17/5 -1)+(-17%5+5)/5
 if (w) os << w << "+"; // w+ to be suppressed if 0
 }
 os << n; if ((n != 0) && (d != 1)) os << "/" << d; // To print the fraction part in both formats
 return os;
}

friend istream& operator>>(istream& is, Fraction_& f) {
 if (f.bMixedFormat_) { // Mixed format support - reads 3 numbers: w, n, d
 cout << "Input fraction in mixed Format" << endl;
 T w, n;
 is >> w >> n >> f.d_;
 f.n_ = w * f.d_ + n;
 }
 else // Simple format support - reads 2 numbers: n, d
 is >> f.n_ >> f.d_;
 *f; // Reduces the fraction
 return is;
}
```



# Fraction: Mixed Format Support in Streaming Operators

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >

Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

- Finally, we put together `Fraction_::Format` class:

```
template<typename T = int> class Fraction_ { public: // ...
private: /* ... */ // Support for Mixed Format
 class Format { private: // Wraps bool so that special IO operators can be defined
 bool bFormat_; // Truthvalue for Format object
 void SetMixedFormat(bool b) const; // Sets Fraction_::bMixedFormat_
 Format(bool b): bFormat_(b) {} // Ctor is private - used only by friend class Fraction_
 friend ostream& operator<<(ostream& os, const Format& m); // Called to set bMixedFormat_
 friend istream& operator>>(istream& is, const Format& m); // Called to set bMixedFormat_
 friend class Fraction_; // Since ctor of Format is private, Fraction_ must be a friend
 };
public:
 // Format markers
 static const Format mixed; // Denotes bMixedFormat_ = true
 static const Format simple; // Denotes bMixedFormat_ = false
 // ...
};
```

- Instantiations of Format markers are:

```
const Fraction::Format Fraction::mixed(true); // Denotes bMixedFormat_ = true
const Fraction::Format Fraction::simple(false); // Denotes bMixedFormat_ = false
```



# Int<N> UDT: Update

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >

Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

NPTEL

# Int<N> UDT: Update



# Int<N> UDT: Update: Agenda

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

```
Fraction <int>
Fraction
<Int<4> >
Poly<Int<4> >
Poly <Fraction
<Int<N> > >
```

Caveat

Tutorial Summary

- We have the following update agenda for Int<N>
  - In the constructor of Int<N>, allow out-of-range values to wrap around instead of assert
  - Implement operator\*(), operator/(), and operator%()



# Int<N>: Constructor with Wraparound

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >

Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

- The constructor of Int<N> is:

```
template<typename T = int, unsigned int N = 4>
class Int_ { public: // ...
 explicit Int_<T, N>(int v = 1): v_(v) { // Two overloads of Constructor
 assert(v_ <= static_cast<int>(MaxInt)); // assert will fire if the value
 assert(v_ >= static_cast<int>(MinInt)); // is out of limits
 } // ...
};
```

- For wraparound, we remove asserts and overload operator\*

```
template<typename T = int, unsigned int N = 4>
class Int_ { public: // ...
 explicit Int_<T, N>(int v = 1): v_(v) // Two overloads of Constructor
 { *(this); }
 Int_<T, N> operator*() { // Wraparound operator
 v_ = v_ % TwoPowerN_T;
 if (v_ > MaxInt_T) v_ -= TwoPowerN_T;
 else if (v_ < MinInt_T) v_ += TwoPowerN_T;
 return *this;
 } // ...
};
```

- With this we get the following wraparound:

```
cout << Int_<>(5) << ' ' << Int_<>(77) << ' ' << Int_<>(-43) << endl; // 5 -3 5
```



# Int<N>: Binary Operators

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >

Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

- With the wraparound, it becomes straightforward to implement binary operators with overflow

```
template<typename T = int, unsigned int N = 4>class Int_ { public: // ...
 friend Int_<T, N> operator+(const Int_<T, N>& i1, const Int_<T, N>& i2) {
 return Int_<T, N>(i1.v_ + i2.v_);
 }
 friend Int_<T, N> operator-(const Int_<T, N>& i1, const Int_<T, N>& i2) {
 return i1 + (-i2); } // return Int_<T, N>(i1.v_ - i2.v_); is also okay
 friend Int_<T, N> operator*(const Int_<T, N>& i1, const Int_<T, N>& i2) {
 return Int_<T, N>(i1.v_ * i2.v_);
 }
 friend Int_<T, N> operator/(const Int_<T, N>& i1, const Int_<T, N>& i2) {
 return Int_<T, N>(i1.v_ / i2.v_);
 }
 friend Int_<T, N> operator%(const Int_<T, N>& i1, const Int_<T, N>& i2) {
 return Int_<T, N>(i1.v_ % i2.v_); } // ...
};
```

- With this we get the following:

```
cout << "Binary Plus: Int(2) + Int(3) = " << (Int(2) + Int(3)) << endl; // 5
cout << "Binary Plus: Int(-6) + Int(-7) = " << (Int(-6) + Int(-7)) << endl; // 3
cout << "Binary Minus: Int(2) - Int(3) = " << (Int(2) - Int(3)) << endl; // -1
cout << "Binary Minus: Int(-6) - Int(-7) = " << (Int(-6) - Int(-7)) << endl; // 1
cout << "Binary Multiply: Int(3) * Int(2) = " << (Int(3) * Int(2)) << endl; // 6
cout << "Binary Multiply: Int(7) * Int(5) = " << (Int(7) * Int(5)) << endl; // 3
cout << "Binary Multiply: Int(-8) * Int(-8) = " << (Int(-8) * Int(-8)) << endl; // 0
cout << "Binary Divide: Int(3) / Int(2) = " << (Int(3) / Int(2)) << endl; // 1
cout << "Binary Divide: Int(7) / Int(-5) = " << (Int(7) / Int(-5)) << endl; // -1
cout << "Binary Residue: Int(3) % Int(2) = " << (Int(3) % Int(2)) << endl; // 1
cout << "Binary Residue: Int(-6) % Int(2) = " << (Int(-6) % Int(2)) << endl; // 0
```



# Int<N>: Binary Operators: Properties

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template  
Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >  
Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

- Try to prove the usual arithmetic properties of the `Int<N>` binary operators for addition, subtraction, multiplication, and division under wraparound:
  - Are all operators *Associative*? For example,
$$\triangleright a + b + c = (a + b) + c = a + (b + c)$$
  - Are addition and multiplication *Commutative*? For example,
$$\triangleright a + b = b + a$$
  - Do multiplication and division *Distribute* over addition and subtraction? For example,
$$\triangleright a * (b + c) = a * b + a * c$$
- Especially, check for the boundary conditions under wraparound:
  - `MaxInt + 1 = MinInt`
  - `MinInt - 1 = MaxInt`
  - `- MinInt = MinInt`
- Consider exception and / or assert support in the constructors and / or operators if some specific values can break the properties



# Mixed UDT Apps

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

NPTEL

## Mixed UDT Apps



# Mixed UDT Apps: Agenda

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators  
Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>  
Fraction  
<Int<4>>  
Poly<Int<4>>  
Poly <Fraction  
<Int<N>>>

Caveat

Tutorial Summary

- We have the following agenda for Mixed UDT Apps
  - Test `Fraction<int>`
  - Test `Fraction<Int<4>>`
  - Test `Poly<int>`: Done in **Tutorial 08**
  - Test `Poly<Fraction<int>>`: Done in **Tutorial 08** - actually using the non-template version of `Fraction`
  - Test `Poly<Int<4>>`
  - Test `Poly<Fraction<Int<4>>>`



# Fraction<int>: Application

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >  
Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

```
#include <iostream>
using namespace std;
#include "Frac.h"

typedef Fraction<int> Fraction;
const Fraction Fraction::UNITY = Fraction(1), Fraction::ZERO = Fraction(0);
bool Fraction::bMixedFormat_ = false;
const Fraction::Format Fraction::mixed(true), Fraction::simple(false);

int main() {
 Fraction fa(5, 3);
 cout << "Fraction fa(5, 3) = " << Fraction::mixed << fa << " = " << Fraction::simple << fa;
 Fraction fb(7, 9);
 cout << "Fraction fb(7, 9) = " << Fraction::mixed << fb << " = " << Fraction::simple << fb;
 cout << "fa + fb = " << Fraction::mixed << (fa + fb) << " = " << Fraction::simple << (fa + fb);
 cout << "fa - fb = " << Fraction::mixed << (fa - fb) << " = " << Fraction::simple << (fa - fb);
 cout << "fa * fb = " << Fraction::mixed << (fa * fb) << " = " << Fraction::simple << (fa * fb);
 cout << "fa / fb = " << Fraction::mixed << (fa / fb) << " = " << Fraction::simple << (fa / fb);
}

Fraction fa(5, 3) = 1+2/3 = 5/3
Fraction fb(7, 9) = 7/9 = 7/9
fa + fb = 2+4/9 = 22/9
fa - fb = 8/9 = 8/9
fa * fb = 1+8/27 = 35/27
fa / fb = 2+1/7 = 15/7
```



# Fraction<Int<4> >: Application

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4> >

Poly<Int<4> >

Poly <Fraction  
<Int<N> > >

Caveat

Tutorial Summary

```
#include <iostream>
using namespace std;
#include "Frac.h"
#include "../Int/Int.h"
typedef Int_<int, 4> Int; typedef Fraction_<Int> Fraction;
const Fraction Fraction::UNITY = Fraction(1), Fraction::ZERO = Fraction(0);
bool Fraction::bMixedFormat_ = false;
const Fraction::Format Fraction::mixed(true), Fraction::simple(false);

int main() {
 Fraction fa(5, 3);
 cout << "Fraction fa(5, 3) = " << Fraction::mixed << fa << " = " << Fraction::simple << fa;
 Fraction fb(7, 10);
 cout << "Fraction fb(7, 10) = " << Fraction::mixed << fb << " = " << Fraction::simple << fb;
 cout << "fa + fb = " << Fraction::mixed << (fa + fb) << " = " << Fraction::simple << (fa + fb);
 cout << "fa - fb = " << Fraction::mixed << (fa - fb) << " = " << Fraction::simple << (fa - fb);
 cout << "fa * fb = " << Fraction::mixed << (fa * fb) << " = " << Fraction::simple << (fa * fb);
 cout << "fa / fb = " << Fraction::mixed << (fa / fb) << " = " << Fraction::simple << (fa / fb);
}
```

```
Fraction fa(5, 3) = 1+2/3 = 5/3
Fraction fb(7, 10) = -2+5/6 = -7/6
fa + fb = 1/2 = 1/2
fa - fb = 1/6 = 1/6
fa * fb = -2+1/2 = -3/2
fa / fb = 2/5 = 2/5
```

Programming in Modern C++



# Poly<Int<4>>: Application

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4>>

Poly<Int<4>>

Poly <Fraction  
<Int<N>>>

Caveat

Tutorial Summary

```
#include <iostream>
using namespace std;
#include "../Int/Int.h"
#include "Poly.h"
typedef Int<int, 4> Int;
const int Int::TwoPowerN_T = 1 << N; // 2^N
const int Int::MaxInt_T = (1 << (N-1))-1; /* 2^(N-1)-1 */ Int::MinInt_T = -(1 << (N-1)); // -2^(N-1)
const Int Int::MaxInt = Int(Int::pow() - 1), Int::MinInt = Int(-Int::pow());

void main() { vector<Int> vf = { 2, 15, 7 };
 Poly<Int> pf1(vf); cout << "pf1(x): " << pf1 << " pf1(2) = " << pf1(2) << endl;
 Poly<Int> pf2; cout << "pf2(x): " << pf2 << " pf2(2) = " << pf2(2) << endl;
 cin >> pf2; /* 3 9 7 2 -11 */ cout << "pf2(x): " << pf2 << " pf2(2) = " << pf2(2) << endl;
 Poly<Int> pf3 = pf1 + pf2; cout << "pf3(x): " << pf3 << " pf3(2) = " << pf3(2) << endl;
 Poly<Int> pf4 = pf1 - pf2; cout << "pf4(x): " << pf4 << " pf4(2) = " << pf4(2) << endl << endl;
}

pf1(x): 7x^2 + -1x^1 + 2. pf1(2) = -4 // 2 = 2, 15 = -1, 7 = 7. pf1(2)= 7*4 + -1*2 + 2 = 28-2+2 = 28 = -4
pf2(x): 1. pf2(2) = 1
Enter degree of the polynomial 3
Enter all the coefficients like a0+a1*x+a2*x^2+....an*x^n
9 7 2 -11 // -7 7 2 5
pf2(x): 5x^3 + 2x^2 + 7x^1 + -7. pf2(2) = 7 // pf2(2)= 5*8 + 2*4 + 7*2 + -7 = 56+8+14-7 = 71 = 71-64 = 7
pf3(x): 5x^3 + -7x^2 + 6x^1 + -5. pf3(2) = 3
pf4(x): -5x^3 + 5x^2 + -8x^1 + -7. pf4(2) = 5
```



# Poly<Fraction<Int<4>>> Application

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT  
friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction  
<Int<4>>

Poly<Int<4>>

Poly <Fraction  
<Int<N>>>

Caveat

Tutorial Summary

```
#include <iostream>
using namespace std;
#include "../Fraction/Frac.h"
#include "../Int/Int.h"
#include "Poly.h"

const int N = 4; typedef Int<int, 4> Int; const int Int::TwoPowerN_T = 1 << N; // 2^N
const int Int::MaxInt_T = (1 << (N-1))-1; /* 2^(N-1)-1 */ Int::MinInt_T = -(1 << (N-1)); // -2^(N-1)
const Int Int::MaxInt = Int(Int::pow() - 1), Int::MinInt = Int(-Int::pow());
typedef Fraction<Int> Fraction; bool Fraction::bMixedFormat_ = false;
const Fraction Fraction::UNITY = Fraction(1), Fraction::ZERO = Fraction(0);
const Fraction::Format Fraction::mixed(true), Fraction::simple(false);
void main() {
 vector<Fraction> vf1 = { Fraction(1, 2), Fraction(-3, 5), Fraction(2, 4) };
 Poly<Fraction> pf1(vf1); cout << "pf1(x): " << pf1 << " pf1(Fraction(2)) = " << pf1(Fraction(2)) << endl;
 vector<Fraction> vf2 = { Fraction(1, 2), Fraction(2, 3) };
 Poly<Fraction> pf2(vf2); cout << "pf2(x): " << pf2 << " pf2(Fraction(2)) = " << pf2(Fraction(2)) << endl;
 Poly<Fraction> pf3 = pf1 + pf2;
 cout << "pf3(x): " << pf3 << " pf3(Fraction(2)) = " << pf3(Fraction(2)) << endl;
 Poly<Fraction> pf4 = pf1 - pf2;
 cout << "pf4(x): " << pf4 << " pf4(Fraction(2)) = " << pf4(Fraction(2)) << endl << endl;
}
pf1(x): 1/2x^2 + -3/5x^1 + 1/2. pf1(Fraction(2)) = 7/6 // pf1(2/1)= 1/2*4 -3/5*2 + 1/2 = 7/6
pf2(x): 2/3x^1 + 1/2. pf2(Fraction(2)) = -5/6 // pf2(2/1) = 2/3*2/1 + 1/2 = 4/3 + 1/2 = 11/6 = -5/6
pf3(x): 1/2x^2 + 1. pf3(Fraction(2)) = 3 // pf4(2/1) = 1/2*4/1 + 1 = 3
pf4(x): 1/2x^2. pf4(Fraction(2)) = 2 // pf4(2/1) = 1/2*4/1 = 2
```



# Caveat: Mixes may fail

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

NPTEL

## Caveat: Mixes may fail



# Caveat in mixing UDTs

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction <Int<4> >

Poly<Int<4> >

Poly <Fraction <Int<N> > >

Caveat

Tutorial Summary

- While `Fraction<int>`, `Poly<int>`, `Int<N>`, or `Poly<Fraction<int> >` work perfectly fine, `Fraction<Int<N> >` or `Poly<Fraction<Int<N> > >` may have some surprise
- This is due to the `T gcd(T, T)` algorithm in the context of `Int<N>`. Normally, we invoke `gcd()` for positive numbers only (that's how the Euler's Algorithm is designed to work)
- However, for `MinInt` in `Int<N>`, we have `-MinInt = MinInt`. Hence, if one of the `gcd()` parameters is `MinInt` we are perpetually in the realm of negative numbers. This leads to an *infinite loop* in the code below:

```
static T gcd(T a, T b) { // Finds the gcd for two +ve integers
 while (a != b) if (a > b) a = a - b; else b = b - a; // N = 4. (-8,3) => (-8,-5) => (-8,3) => ...
 return a;
}
```

- So we choose to `throw` (and eventually `assert` in the constructor) when one of the `gcd()` arguments is negative (eventually `MinInt`)

```
static T gcd(T a, T b) { // Finds the gcd for two +ve integers
 if (a < 0) throw "Negative first arg in gcd";
 if (b < 0) throw "Negative second arg in gcd";
 while (a != b) if (a > b) a = a - b; else b = b - a; // For N = 4, a = -8 is an infinite loop
 return a;
}
```

- How to fix?



# Tutorial Summary

Tutorial T09

Partha Pratim  
Das

Tutorial Recap

Objective &  
Outline

Update UDTs

Fraction UDT

friend Operators

Template

Mixed Format

Int<N> UDT

Wraparound

Binary Ops

Mixed UDT Apps

Fraction <int>

Fraction

<Int<4> >

Poly<Int<4> >

Poly <Fraction

<Int<N> > >

Caveat

Tutorial Summary

- UDTs Fraction, Int and Poly have been updated with various features
- Mixed applications involving multiple UDTs have been checked
- Caveat or loophole has been identified

NPTEL



Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

# Programming in Modern C++

Tutorial T10: How to optimize C++11 programs using Rvalue and Move Semantics?

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 Objectives

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- To understand optimization by copy elision
- To understand copy / move optimization by Rvalues and Move Semantics

NPTEL



# Tutorial Outline

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

## 1 Optimizing C++11 Programs

## 2 Copy Elision

- Copy Initialization
- Return Value Optimization (RVO)
- Language Specification

## 3 Sorting Objects

- Copy Support
- Statistics Support
- Move Support
- Summary
- Project Codes
- Problems

## 4 Tutorial Summary

NPTEL



# Optimizing C++11 Programs

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Optimizing C++11 Programs

### Sources:

- [Move semantics and rvalue references in C++11](#), cprogramming, Alex Allain, 2019
- [Copy elision](#), wikipedia



# Optimizing C++11 Programs

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- C++ has always produced fast programs
- Unfortunately, until C++11, there has been an obstinate wart that slows down many C++ programs:
  - the *creation of temporary objects*
- Sometimes these temporary objects can be optimized away by the compiler by *copy elision*<sup>1</sup> (the return value optimization, for example). But this is not always the case, and it can result in expensive object copies
- Copy elision (or omission) depends primarily on identification of **rvalues** by the compiler and can be optimized away
- In addition to what the compiler can do, we can reduce copies by explicitly marking **rvalues** in the code by Rvalue references and by providing the move operations along with the copy operations (if needed)
- We first elucidate some common scenarios of copy elision that the language standard specifies and the compiler exploits for optimization
- Next we show through a small sorting project how the programmer can expose good move opportunities for the compiler to optimize copies

---

<sup>1</sup>compiler optimization technique that eliminates unnecessary copying of objects



# Copy Elision

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

**Copy Elision**

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Copy Elision

### Sources:

- [Copy elision, cppreference](#)
- [Copy elision in C++, geeksforgeeks, 2017](#)
- [Copy elision, wikipedia](#)



# Copy Elision

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- In C++ programming, **copy elision**<sup>2</sup> refers to a *compiler optimization technique that eliminates unnecessary copying of objects*
- The C++ language standard generally allows implementations to perform any optimization, provided the resulting program's observable behavior is the same as if, that is pretending, the program were executed exactly as mandated by the standard.
- Beyond that, the standard also describes a few situations where copying can be eliminated even if this would alter the program's behavior
  - the most common being the *return value optimization*
  - Another widely implemented optimization, described in the C++ standard, is when a temporary object of class type is copied to an object of the same type. As a result
    - ▷ *copy-initialization* is usually equivalent to *direct-initialization* in terms of performance, but semantically,
    - ▷ copy-initialization still requires an accessible copy constructor
- The optimization cannot be applied to a temporary object that has been bound to a reference

---

<sup>2</sup> *elision* is the omission of a sound or syllable when speaking (as in I'm, let's)



# Copy Elision: Copy Initialization

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Copy Elision: Copy Initialization



# Copy Elision: Copy Initialization

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- What will be the output?

```
#include <iostream>
int n = 0;
struct C {
 explicit C(int) { }
 C(const C&) { ++n; } // the copy constructor has a visible side effect
 // it modifies an object with static storage duration
};
int main() {
 C c1(42); // direct-initialization, calls C(int). c1 is lvalue
 C c2 = C(77); // copy-initialization. C(77) by C(int) is rvalue, skips C(const C&)
 std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
}
```

- Interestingly both GCC-C++ and MSVC++ print 0 even in debug build
- Copy constructor `C::C(const C&)` is not even invoked
- If you think this is because `C::C(const C&)` does not do anything meaningful for the object, check the next version



# Copy Elision: Copy Initialization

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- What will be the output?

```
#include <iostream>
int n = 0;
struct C { int i;
 explicit C(int i) : i(i) { std::cout << i << ' '; }
 C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
 ~C() { std::cout << "~" << i << ' '; }
};
int main() {
 C c1(42); // direct-init., calls C(int). c1 is lvalue // 42
 C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77
 std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
} // ~77 ~42
```

- `C::C(const C&)` is just not invoked!
- Yet, if you comment the copy constructor and explicitly delete it (`C(const C&) = delete;`) so that no free copy constructor is provided, C++11 will give **error: use of deleted function ‘C::C(const C&)**



# Copy Elision: Copy Initialization

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Let us construct an object from an **lvalue**

```
#include <iostream>
int n = 0;
struct C { int i;
 explicit C(int i) : i(i) { std::cout << i << ' '; }
 C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
 ~C() { std::cout << "~" << i << ' '; }
};
int main() {
 C c1(42); // direct-init., calls C(int). c1 is lvalue // 42
 C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77
 C c3 = c1; // copy-init., calls C(const C&) as c1 is lvalue // 43
 std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 1
} // ~43 ~77 ~42
```



# Copy Elision: Copy Initialization

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Using `-fno-elide-constructors` option to disable copy-elision:

```
#include <iostream>
int n = 0;
struct C { int i;
 explicit C(int i) : i(i) { std::cout << i << ' '; }
 C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
 ~C() { std::cout << "~" << i << ' '; }
};
int main() {
 C c1(42); // direct-init., calls C(int). c1 is lvalue // 42
 C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77 78 ~77
 C c3 = c1; // copy-init., calls C(const C&) as c1 is lvalue // 43
 std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 2
} // ~43 ~78 ~42
```



# Copy Elision: Return Value Optimization (RVO)

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Copy Elision: Return Value Optimization (RVO)



# Copy Elision: Return Value Optimization (RVO)

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Similar behaviour would be observed through function return by direct construction:

```
#include <iostream>
int n = 0;
struct C { int i;
 explicit C(int i) : i(i) { std::cout << i << ' '; }
 C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
 ~C() { std::cout << "~" << i << ' '; }
};
C f(int i) {
 return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is to be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
 C c(i); // directly constructed object: c is lvalue needs C(int)
 return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
 f(19); // f(19) is rvalue - unused and destructed // 19 ~19
 g(35); // f(19) is rvalue - unused and destructed // 35 ~35
 std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
}
```



# Copy Elision: Return Value Optimization (RVO)

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Similar behaviour is also observed if the return value is used in initialization without being discarded – however, the destruction order changes:

```
#include <iostream>
int n = 0;
struct C { int i;
 explicit C(int i) : i(i) { std::cout << i << ' '; }
 C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
 ~C() { std::cout << "~" << i << ' '; }
};
C f(int i) {
 return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is to be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
 C c(i); // directly constructed object: c is lvalue needs C(int)
 return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
 C c1 = f(19); // copy-init. f(19) by C(int) is rvalue, skips C(const C&) // 19
 C c2 = g(35); // copy-init. g(35) by C(int) is rvalue, skips C(const C&) // 35
 std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
} // ~35 ~19
```



# Copy Elision: Return Value Optimization (RVO)

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Using `-fno-elide-constructors` option to disable copy-elision:

```
#include <iostream>
int n = 0;
struct C { int i;
 explicit C(int i) : i(i) { std::cout << i << ' '; }
 C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
 ~C() { std::cout << "~" << i << ' '; }
};
C f(int i) {
 return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is to be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
 C c(i); // directly constructed object: c is lvalue needs C(int)
 return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
 C c1 = f(19); // copy-init. f(19) by C(int) is rvalue, skips C(const C&) // 19 20 ~19 21
 C c2 = g(35); // copy-init. g(35) by C(int) is rvalue, skips C(const C&) // 35 36 ~35 37
 std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 4
} // ~37 ~21
```



# Copy Elision: Language Specification

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Copy Elision: Language Specification

### Sources:

- [Copy elision](#), cplusplus.com



# Mandatory Elision: Copy / Move Operations

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Under the following circumstances, the compilers are required to omit the copy and move construction of class objects, even if the copy / move constructor and the destructor have observable side-effects
- Objects are constructed directly into the storage where they would be copied / moved to*
- The copy / move constructors need not be present or accessible:*
  - In a return statement, when the operand is a **prvalue** of the same class type (ignoring cv-qualification) as the return type:

```
T f() {
 return T();
}
```

```
f(); // only one call to default constructor of T
```

- More are specified for **C++17**



# Non-Mandatory Elision: Copy / Move Operations

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Under the following circumstances, the compilers are *permitted, but not required* to omit the copy and move construction of class objects, *even if the copy / move constructor and the destructor have observable side-effects*
- Objects are constructed directly into the storage where they would be copied / moved to*
- This is an optimization:** *even when it takes place and the copy / move constructor is not called, it still must be present and accessible (as if no optimization happened at all), otherwise the program is ill-formed:*
  - In a *return statement*, when the operand is a named object (and not a function or a catch clause param) with automatic storage duration, and which is of the same class type (ignoring cv-qualification) as the function return type. This variant of copy elision is known as **Named Return Value Optimization (NRVO)**
  - In the *initialization of an object*, when the source object is a nameless temporary and is of the same class type (ignoring cv-qualification) as the target object. When the nameless temporary is the operand of a return statement, this variant of copy elision is known as **Return Value Optimization (RVO)**
  - Return value optimization is mandatory and no longer considered as copy elision since **C++17**



# Sorting Objects

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Sorting Objects



# Sorting Objects

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- To illustrate the effect by copy optimization, we consider a tiny sorting project
- We intend to sort objects of a data class **D** having resource of a class **R**
- We define the following to get started:
  - Resource class **R**
  - Data class **D**
  - A template function **swap**
  - A template function **sort** to bubble sort an array
  - The **main** function to initialize an array and sort it
- We are interested to see the trade-off of move and copy. So we build a statistics support in the code to count the number of constructions and destructions of the resource objects from class **R**
- Initial version works with only Copy operations
- We next add move operations in Data class **D** and move support in **swap** function
- We compare the statistics to show the huge benefit accrued with the move semantics



# Sorting Objects: Copy Support

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Sorting Objects: Copy Support



# Resource Class, R

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Let us consider a resource class **R**, and
- A data class **D** having resource **R**:

```
struct R { // Resource class - wrapper of int
 int i; // Wrapped resource
 R(int i) : i(i) {} // Parametric constructor
 R(const R& r) : i(r.i) {} // Copy constructor
 ~R() {} // Destructor
};

struct D { // Data class with resource
 R* r; // Resource to be dynamically constructed / destructed
 // ...
};
```



# Data Class, D

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

```
struct D { // Data class with resource
 R* r; // Resource to be dynamically constr. / destru.
 D() : r(nullptr) { } // Default constructor - null resource
 D(int i) : r(new R(i)) { } // Parametric constructor - create resource
 D(const D& d) : r(new R(*(d.r))) { } // Copy constructor - copy resource
 D& operator=(const D& d) {
 if (this != &d) { // Self copy guard
 delete r; // Free resource
 r = new R(*(d.r)); // Copy resource
 }
 return *this;
 }
 ~D() { delete r; } // Destructor - free resource
 friend bool operator>(const D& c1, const D& c2) { // Compare D objects for sorting
 return c1.r->i > c2.r->i;
 }
 friend std::ostream& operator<<(std::ostream& os, const D& d) { // Stream D objects
 os << d.r->i << ' ';
 return os;
 }
};
```



# sort Function

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- We store **N** number of **D** objs in an array
- We sort the array by Bubble Sort in ascending order

```
template<typename T>
void swap(T& a, T&b) { // Swap a and b using copy
 T t = a; // t copy-created from a: two a's
 a = b; // a copy-assigned from b: two b's, one a destroyed
 b = t; // b copy-assigned from t: two t's, one b destroyed
} // t destroyed

template<typename T>
void sort(T arr[], int n) { // Bubble Sort for easy analysis
 for (int i = 0; i < n - 1; ++i)
 for (int j = 0; j < n - i - 1; ++j)
 if (arr[j] > arr[j + 1]) { // Compare by D::operator>
 swap(arr[j], arr[j + 1]); // 3 constr.s and destr.s of R objs with copy
 // 0 constr. and destr. of R objs with move
 }
}
```



# main Function

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

```
int main() { // To populate and sort an array of D objs having R obj resources
 const int N = 10; // Size of array and number of elements
 D arr[N]; // Defa. initialization of array - use D::D() calls N times

 // Assignments of array elements with D objs having R obj resources
 // Fill with a strictly decreasing sequence for worst case of Bubble Sort
 for (int i = N - 1; i >= 0; --i)
 arr[i] = D(N - i); // Construct by D::D(int), assign by D::operator=(const D&)
 // construct / destruct R objs
 for (int i = 0; i < N; ++i) // Print array before sorting. 10 9 8 7 6 5 4 3 2 1
 std::cout << arr[i]; std::cout << std::endl;
 sort(arr, N); // Sort array in ascending order
 for (int i = 0; i < N; ++i) // Print array after sorting. 1 2 3 4 5 6 7 8 9 10
 std::cout << arr[i]; std::cout << std::endl;
}
```

- To get an estimate for the resource construct. and destruct., we build a worst-case for Bubble Sort, that is, populate `arr` in strictly descending order. Being sorting, this is dominated by `swaps`.
- Clearly in the worst case number of  $\text{swaps} = \sum_{i=1}^{N-1} i = \frac{N*(N-1)}{2}$ . Hence number of (unnecessary) resource constructions and destructions =  $3 * \# \text{ of swaps} = \frac{3*N*(N-1)}{2}$



# Sorting Objects: Statistics Support

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Sorting Objects: Statistics Support



# Resource Class R with Statistics

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- To count the exact number of constructions and destructions of `R` objects, we add three static counters in `R`
- We also add a static method `stat()` to print the statistics at anytime from anywhere

```
struct R { // Resource class
 int i; // Wrapped resource
 R(int i) : i(i) { ++nCtor; } // Parametric constructor
 R(const R& r) : i(r.i) { ++nC_Ctor; } // Copy constructor
 ~R() { ++nDtor; } // Destructor

 static unsigned int nCtor; // Count of direct construction of R objects
 static unsigned int nC_Ctor; // Count of copy construction of R objects
 static unsigned int nDtor; // Count of destruction of R objects

 static void stat(std::string s) { // Print R object statistics
 std::cout << s /* Banner message */ << "R obj Created = " << R::nCtor <<
 " R obj Copy Created = " << R::nC_Ctor << " R obj Destroyed = " << R::nDtor <<
 std::endl;
 }
};
```



# Resource Class R with Statistics

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Static counters of `R` are globally instantiated and initialized with 0's.
- We also add helper class `Stat` whose constructor and destructor calls `R::stat()`. Next we globally instantiate an object `extremeStat` of `Stat`
  - Being global static, `extremeStat` is constructed before `main()` is called and is destructed after `main()` returns
  - Hence the statistics are printed before calling `main()` and after returning from `main()`

```
// Instantiations of static R objects in global namespace
unsigned int R::nCtor = 0; // Count of direct construction of R objects
unsigned int R::nC_Ctor = 0; // Count of copy construction of R objects
unsigned int R::nDtor = 0; // Count of destruction of R objects

struct Stat { // Helper class to print R objects statistics
 Stat() { R::stat("Program Start: "); } // Construct before main(), initial stat
 ~Stat() { R::stat("Program End: "); } // Destruct after main(), final stat
} extremeStat;
```



# main Function with Statistics

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

```
int main() { // To populate and sort an array of D objs having R obj resources
 const int N = 10; // Size of array and number of elements
 D arr[N]; // Defa. initialization of array - use D::D() calls N times
 R::stat("Array Defa: "); // Statistics after Defa. initialization of array
 // Assignments of array elements with D objs having R obj resources
 // Fill with a strictly decreasing sequence for worst case of Bubble Sort
 for (int i = N - 1; i >= 0; --i)
 arr[i] = D(N - i); // Construct by D::D(int), assign by D::operator=(const D&)
 // construct / destruct R objs
 R::stat("Array Init: "); // Statistics after assignment of array
 for (int i = 0; i < N; ++i) // Print array before sorting
 std::cout << arr[i]; std::cout << std::endl;
 sort(arr, N); // Sort array in ascending order
 R::stat("Array Sort: "); // Statistics after sorting of array
 for (int i = 0; i < N; ++i) // Print array after sorting
 std::cout << arr[i]; std::cout << std::endl;
} // Statistics after destruction of array elements by s.^Stat()
```

Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0

Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0

Array Init: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0

10 9 8 7 6 5 4 3 2 1

Array Sort: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 135

1 2 3 4 5 6 7 8 9 10

Program End: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 145



# Analysis of Statistics

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision  
Copy Initialization  
Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- **Program Start:** R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
  - Static object `extremeStat` constructed by `Stat::Stat()` before `main()` is invoked, reports statistics
- **Array Defa:** R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
  - `D arr[N]: N = 10` D objects are constructed by `D::D()`. As `D::r` is set to `nullptr` in each, no R object is constructed
- **Array Init:** R obj Created = 10 R obj Copy Created = 10 R obj Destroyed = 10
  - ... = `D(N - i): N = 10` D objects are constructed by `D::D(int)`. As `D::r` is set to `new R(i)` in each, `N = 10` R object is constructed
  - `arr[i] = ...: D(N - i)` is now *copy assigned* to `arr` elements by `D::D(const D& d)`. Hence, the resource R objects is destructed (`delete r`) and constructed (`new R(*(d.r))`) for each
  - Note that `D(N - i)` is an *rvalue*, yet it is *copy assigned* as there is no *move assignment*
- 10 9 8 7 6 5 4 3 2 1
  - `arr` before sorting. Filled with a strictly decreasing sequence
- **Array Sort:** R obj Created = 10 R obj Copy Created = 145 R obj Destroyed = 145
  - `sort(arr, N);:` Being the worst case of bubble sort,  $\frac{3*N*(N-1)}{2} = \frac{3*10*(10-1)}{2} = 135$  R objects are constructed by `R::R(const R&)` and destructed by `R::~R()` for 45 swaps. Note that `t`, `a`, and `b` in `swap` are *lvalues*
- 1 2 3 4 5 6 7 8 9 10
  - `arr` after sorting in increasing order
- **Program End:** R obj Created = 10 R obj Copy Created = 145 R obj Destroyed = 155
  - `int main() { ... }:` Remaining `N = 10` D objects destructed by `D:: D()` with `delete D::r`. Static object `extremeStat` destructed by `Stat::~Stat()` after `main()` returns reports



# Sorting Objects: Move Support

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Sorting Objects: Move Support



# Data Class D with Move Support

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- To minimize copies, we provide move operations in class **D** to be able to move **rvalues** whenever possible

```
struct D { // Data class with resource
 R* r; // Resource to be dynamically constru. / destru.
 D(); // Default constructor - null resource
 D(int i); // Parametric constructor - create resource
 D(const D& d); // Copy constructor - copy resource
 D& operator=(const D& d); // Copy assignment - copy resource
 ~D(); // Destructor - free resource

 D(D&& d) : r(d.r) { d.r = nullptr; } // Move constructor - move resource, ownership
 D& operator=(D&& d) { // Move assignment - move resource, ownership
 if (this != &d) { // Self move guard
 r = d.r; // Move resource
 d.r = nullptr; // Take ownership
 }
 return *this;
 }
};
```

- We again run the program and gather statistics



# Analysis of Statistics: Move Support in Class D

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Here is the statistics with move support. We note the changes:
- Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
- Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
- Array Init: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0
  - ... = D(N - i): N = 10 D objects are constructed by `D::D(int)`. As `D::r` is set to `new R(i)` in each, N = 10 R object is constructed
  - `arr[i] = ...`: `D(N - i)` is now *move assigned* to `arr` elements by `D::D(D&& d)` since `D(N - i)` is a `rvalue`
  - Hence, no resource R object is destructed or constructed - just ownership of resource is transferred
- 10 9 8 7 6 5 4 3 2 1
- Array Sort: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 135
- 1 2 3 4 5 6 7 8 9 10
- Program End: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 145



# swap Function with Move Support

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- To minimize copies further, we provide move support in `swap()` function using `std::move`

```
template<typename T>
void swap(T& a, T&b) { // Swap a and b using move
 T t = std::move(a); // t move-created from a: a's ownership transferred to t
 a = std::move(b); // a move-assigned from b: b's ownership transferred to a
 b = std::move(t); // b move-assigned from t: t's ownership transferred to b
} // t destroyed, but no resource destruction as t had no ownership
```



# Analysis of Statistics: Move Support in Class D and Function swap

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Here is the statistics with move support. We note the changes:
- Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
- Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
- Array Init: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0
- 10 9 8 7 6 5 4 3 2 1
- Array Sort: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0
  - `sort(arr, N);`: Being the worst case of bubble sort,  $\frac{N*(N-1)}{2} = \frac{10*(10-1)}{2} = 45$  swaps are performed. But no swap copies any R object **only moves**
  - Hence no unnecessary construction and destruction of R objects
- 1 2 3 4 5 6 7 8 9 10
- Program End: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 10



# Sorting Objects: Summary

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Sorting Objects: Summary



# Analysis of Statistics: Summary

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

| struct D  |           | void swap(T&, T&) |      | R(int) | R(const R&)              | $\sim R()$                                     |
|-----------|-----------|-------------------|------|--------|--------------------------|------------------------------------------------|
| Only Copy | Copy+Move | Copy              | Move |        |                          |                                                |
| Yes       |           | Yes               |      | N      | $\frac{3N*(N-1)}{2} + N$ | $\frac{3N*(N-1)}{2} + 2N = \frac{N*(3N+1)}{2}$ |
| Yes       |           |                   | Yes  | N      | $\frac{3N*(N-1)}{2} + N$ | $\frac{3N*(N-1)}{2} + 2N = \frac{N*(3N+1)}{2}$ |
|           | Yes       | Yes               |      | N      | $\frac{3N*(N-1)}{2}$     | $\frac{3N*(N-1)}{2} + N = \frac{N*(3N-1)}{2}$  |
|           | Yes       |                   | Yes  | N      | 0                        | N                                              |

- With move support in the class and in swap function, we can elide  $O(N^2)$  copies (and destructions)



# Sorting Objects: Project Codes

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

NPTEL

## Sorting Objects: Project Codes



# Resource Class R

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

```
struct R { // Resource class
 int i; // Wrapped resource
 R(int i) : i(i) { ++nCtor; } // Parametric constructor
 R(const R& r) : i(r.i) { ++nC_Ctor; } // Copy constructor
 ~R() { ++nDtor; } // Destructor

 static unsigned int nCtor; // Count of direct construction of R objects
 static unsigned int nC_Ctor; // Count of copy construction of R objects
 static unsigned int nDtor; // Count of destruction of R objects

 static void stat(std::string s) { // Print R object statistics
 std::cout << s /* Banner message */ << "R obj Created = " << R::nCtor <<
 " R obj Copy Created = " << R::nC_Ctor << " R obj Destroyed = " << R::nDtor <<
 std::endl;
 }
};

// Instantiations of static R objects in global namespace
unsigned int R::nCtor = 0; // Count of direct construction of R objects
unsigned int R::nC_Ctor = 0; // Count of copy construction of R objects
unsigned int R::nDtor = 0; // Count of destruction of R objects

struct Stat { // Helper class to print R objects statistics
 Stat() { R::stat("Program Start: "); } // Construct before main(), initial stat
 ~Stat() { R::stat("Program End: "); } // Destruct after main(), final stat
} extremeStat;
```



# Data Class D

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

```
struct D { // Data class with resource
 R* r; // Resource to be dynamically constru. / destru.
 D() : r(nullptr) {} // Default constructor - null resource
 D(int i) : r(new R(i)) {} // Parametric constructor - create resource
 D(const D& d) : r(new R(*(d.r))) {} // Copy constructor - copy resource
 D& operator=(const D& d) { // Copy assignment - copy resource
 if (this != &d) { // Self copy guard
 delete r; /* Free resource */ r = new R(*(d.r)); // Copy resource
 } return *this;
 }
 ~D() { delete r; } // Destructor - free resource
#ifndef __MOVE__ // If __MOVE__ is defined (set -D=__MOVE__ flag in GCC to define __MOVE__), use move operations
 D(D&& d) : r(d.r) { d.r = nullptr; } // Move constructor - move resource, ownership
 D& operator=(D&& d) { // Move assignment - move resource, ownership
 if (this != &d) { // Self move guard
 r = d.r; /* Move resource */ d.r = nullptr; // Take ownership
 } return *this;
 }
#endif // __MOVE__ // End of conditional compilation by __MOVE__
 friend bool operator>(const D& c1, const D& c2) { // Compare D objects for sorting
 return c1.r->i > c2.r->i;
 }
 friend std::ostream& operator<<(std::ostream& os, const D& d) { // Stream D objects
 os << d.r->i << ' '; return os;
 }
}
```



# swap & sort Functions

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

```
#ifndef _MOVE_ // If _MOVE_ is not defined, use copy version
template<typename T>
void swap(T& a, T&b) { // Swap a and b using copy
 T t = a; // t copy-created from a: two a's
 a = b; // a copy-assigned from b: two b's, one a destroyed
 b = t; // b copy-assigned from t: two t's, one b destroyed
} // t destroyed
#else // If _MOVE_ is defined (set -D=_MOVE_ flag in GCC to define _MOVE_), use move version
template<typename T>
void swap(T& a, T&b) { // Swap a and b using move
 T t = std::move(a); // t move-created from a: a's ownership transferred to t
 a = std::move(b); // a move-assigned from b: b's ownership transferred to a
 b = std::move(t); // b move-assigned from t: t's ownership transferred to b
} // t destroyed, but no resource destruction as t had no ownership
#endif // _MOVE_ // End of conditional compilation by _MOVE_

template<typename T>
void sort(T arr[], int n) { // Bubble Sort for easy analysis
 for (int i = 0; i < n - 1; ++i)
 for (int j = 0; j < n - i - 1; ++j)
 if (arr[j] > arr[j + 1]) { // Compare by D::operator>()
 swap(arr[j], arr[j + 1]); // 3 constr.s and destr.s of R objs with copy
 // 0 constr. and destr. of R objs with move
 }
}
```



# main Function

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value  
Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

```
int main() { // To populate and sort an array of D objs having R obj resources
 const int N = 10; // Size of array and number of elements
 D arr[N]; // Defa. initialization of array - use D::D() calls N times
 R::stat("Array Defa: "); // Statistics after Defa. initialization of array

 // Assignments of array elements with D objs having R obj resources
 // Fill with a strictly decreasing sequence for worst case of Bubble Sort
 for (int i = N - 1; i >= 0; --i)
 arr[i] = D(N - i); // Construct by D::D(int), assign by
 // D::operator=(const D&) for copy, constr. / destru. R objs
 // D::operator=(D&&) for move, no constr. / destru. R objs

 R::stat("Array Init: "); // Statistics after assignment of array
 for (int i = 0; i < N; ++i) // Print array before sorting
 std::cout << arr[i];
 std::cout << std::endl;

 sort(arr, N); // Sort array in ascending order

 R::stat("Array Sort: "); // Statistics after sorting of array
 for (int i = 0; i < N; ++i) // Print array after sorting
 std::cout << arr[i];
 std::cout << std::endl;
} // Statistics after destruction of array elements by s.^Stat()
```



# Problems

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Provide construction / destruction counting and statistics generation support for class **D**
- Consider that the resource in **D** is held as a data member (**R r;**) and not as a pointer (**R \*r;**). Provide appropriate support in classes **R** and **D** to avoid unnecessary copies during sorting
- Explore the move support in standard library containers, especially **vector** and **map**



# Tutorial Summary

Tutorial T10

Partha Pratim  
Das

Objective &  
Outline

Optimizing  
C++11 Programs

Copy Elision

Copy Initialization

Return Value

Optimization (RVO)

Language  
Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Understood optimization by copy elision
- Understood copy / move optimization by Rvalues and Move Semantics
- Developed a complete sorting project with copy optimization by move

NPTEL



Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

# Programming in Modern C++

Tutorial T11: Compatibility of C and C++: Part 1: Significant Features

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 Objectives

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++  
void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- We often say that “C is a subset of C++”. It is far from truth. There are various intra-dialect incompatibilities in C ([C89](#), [C99](#), [C11](#)), in C++ ([C++03](#), [C++11](#), [C++14](#), [C++17](#), ...), and inter-dialect incompatibilities across languages. We need to understand these differences and their effect in the programs we write
- We take a look at the C/C++ communities and consider views of different sections of communities to understand why we need the compatibility - at least, the clear understanding for it
- We discuss the major compatibility issues between C and C++. To keep the discussion manageable, we primarily focus between [C99](#) and [C++11](#)
- We also discuss the workarounds to write more compatible code between C and C++



# Tutorial Outline

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

## 1 Why is Compatibility of C and C++ important?

## 2 Compatibility of C and C++

- void\*
- const
- enum
- ODR
- void Param
- Nested struct
- VLA
- FAM
- restrict

## 3 Tutorial Summary

NPTEL



# Why is Compatibility of C and C++ important?

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

NPTEL

## Why is Compatibility of C and C++ important?

**Source:** The C/C++ Users Journal, Jul-Aug-Sep, 2002. Accessed 15-Sep-21

**C and C++: Siblings,** B. Stroustrup

**C and C++: A Case for Compatibility,** B. Stroustrup

**C and C++: Case Studies in Compatibility,** B. Stroustrup



# Why is Compatibility of C and C++ important?

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- The *C and C++ programming languages are closely related* but have *many significant differences*
- **There is no C/C++ language, but there is a C/C++ community.** Millions of programmers and organization who use C and/or C++ form the community comprising three major groups:
  - *Programmers who use C only:* Especially the embedded systems community. Many programmers working with C programs that never call a C++ library. However, most (?) C programmers occasionally use C++ directly and many rely on C++ libraries. Hence, the C programmer must be aware of C++ in the same way as a C++ programmer must be aware of C.
  - *Programmers who use C++ only:* Is it possible? Most programmers would need to call a C library. Hence, the programmer needs to understand the constructs in its header files - use of malloc() rather than new, the use of arrays rather than C++ standard library containers, and the absence of exception handling. So all C++ programmers are C programmers.
  - *Programmers who use both C and C++:*
- Compatibility maximizes the community of contributors. Each dialect and incompatibility limits the
  - market for vendors/suppliers/builders
  - set of libraries and tools for users - single product (IDE, compiler, analyzer, etc.) for both languages
  - set of collaborators (suitable employees, students, consultants, experts, etc.) for projects



# Why is Compatibility of C and C++ important?

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*  
const  
enum  
ODR  
void Param

Nested struct  
VLA  
FAM  
restrict

Tutorial Summary

- Bjarne Stroustrup, the creator of C++, has suggested that *the incompatibilities between C and C++ should be reduced as much as possible in order to maximize interoperability between the two languages*
- Others argue that C and C++ are two different languages - *compatibility between them is useful but not vital*; and efforts to reduce incompatibility *should not hinder improvement of each language in isolation*
- C99 “endorse[d] the principle of maintaining the largest common subset” between C and C++ “while maintaining a distinction between them and allowing them to evolve separately”, and stated that the authors were “content to let C++ be the big and ambitious language”
- Several additions of C99 are *not supported in the current C++ standard or conflicted with C++ features*, such as *variable-length arrays*, native *complex number* types and the *restrict* type qualifier
- On the other hand, C99 *reduced some other incompatibilities compared with C89* by incorporating C++ features such as *// comments* and *mixed declarations and code*



# Compatibility of C and C++

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

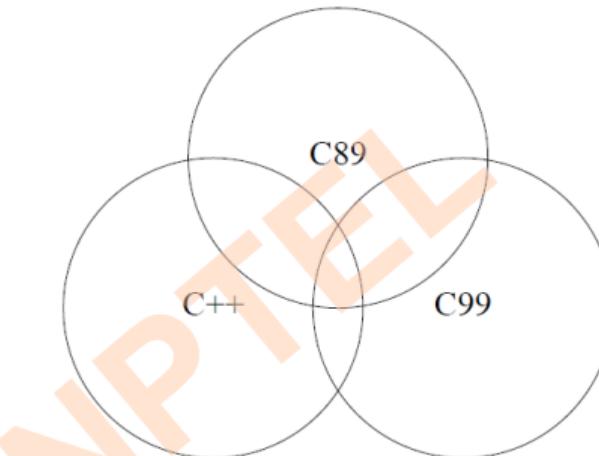
Nested struct

VLA

FAM

restrict

Tutorial Summary



*Feature compatibility between C++98, C89, and C99. "There are features in all 7 areas" -  
C and C++: Siblings, B. Stroustrup, 2002*

## Compatibility of C and C++

Source: Accessed 15-Sep-21

C and C++: A Case for Compatibility, B. Stroustrup

C and C++: Case Studies in Compatibility, B. Stroustrup

Compatibility of C and C++, HandWiki

Compatibility of C and C++, Wikipedia

Annex C.1 of the ISO C++ standard

Programming in Modern C++

Partha Pratim Das

T11.7



# Compatibility of C and C++

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*  
const  
enum  
ODR

void Param  
Nested struct

VLA

FAM

restrict

Tutorial Summary

- *C is not a subset of C++, and nontrivial C programs will not compile as C++ code without change*
- *Likewise, C++ introduces many features that are not available in C and in practice almost all code written in C++ is not conforming C code*
- Here, we focus on differences that cause **conforming C code to be ill-formed C++ code**, or to be **conforming / well-formed in both languages but to behave differently in C and C++**
- We take following approach for the discussions:
  - To explain the compatibility, incompatibility and work-around in an understandable way, we write the same code in `main.c` and `main.cpp` and compile with `gcc` to get the language specific behavior
  - We also use dialect specific `-std` flags wherever relevant. Most comparisons are done with respect to **C99** and **C++11**
  - We present the compiler messages and / or output to elucidate the effects
  - We present a summary of the compatibility issues at the end in comparative tabular form



# Compatibility of C and C++: void\*

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*  
const  
enum  
ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- One commonly encountered difference is C being more **weakly-typed** regarding pointers
- Specifically, C allows a void\* pointer to be assigned to any pointer type without a cast, while C++ does not**
- This idiom appears often in C code using `malloc` memory allocation, or in the passing of context pointers to the `POSIX pthreads API`, and other frameworks involving `callbacks`
- For example, the following is valid in C but not C++:

```
void *ptr;
/* Implicit conversion from void* to int* */
int *i = ptr;
```

or similarly:

```
int *j = malloc(5 * sizeof *j); /* Implicit conversion from void* to int* */
```

In order to make the code compile as both C and C++, one must use an **explicit cast**, as follows (with some caveats in both languages)

```
void *ptr;
int *i = (int *)ptr;
int *j = (int *)malloc(5 * sizeof *j);
```



# Compatibility of C and C++: const

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- C++ is also more strict than C about pointer assignments that discard a `const` qualifier. For example, assigning a `const int*` value to an `int*` variable:

```
int main() { const int* p = 0;
 int* q = p; // const qualifier being discarded
 }
```

In C++, this is invalid and generates a compiler error (unless an explicit typecast is used), while in C this is allowed (although many compilers emit a warning)

```
$ gcc main.cpp main.c
```

```
main.cpp:3:14: error: invalid conversion from 'const int*' to 'int*' [-fpermissive]
 int* q = p;
```

```
main.c:3:14: warning: initialization discards 'const' qualifier from pointer target type
 [-Wdiscarded-qualifiers]
 int* q = p;
```

- In C++ a `const` variable must be initialized; in C this is not necessary. For

```
int main() { const int i = 5;
 const int j; // const variable not initialized
 }
$ gcc main.cpp main.c
```

```
main.cpp:12:15: error: uninitialized const 'j' [-fpermissive]
 const int j;
```



# Compatibility of C and C++: string.h and enum

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++  
void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- C++ changes some C standard library functions to add additional overloaded functions with `const` type qualifiers, for example, consider `strchr()` function in `string.h` in C and `cstring` in C++

```
// string.h
char *strchr(const char *str, int character)
// cstring
const char *strchr(const char * str, int character);
char *strchr (char * str, int character);
```

So when a C file is compiled with C++ compiler different calls to `strchr()` may bind to different overloads in C++

- C++ is also more strict in conversions to `enums`: `ints` cannot be implicitly converted to `enums` as in C.

```
enum week { Mon, Tue, Wed, Thur, Fri, Sat, Sun };
int main() { enum week day;
 int dayindex = 2;
 day = dayindex;
}

$ gcc main.c main.cpp
```

```
main.cpp:23:11: error: invalid conversion from 'int' to 'week' [-fpermissive]
 day = dayindex;
```

- Also, Enumeration constants (`enum` enumerators) are always of type `int` in C, whereas they are distinct types in C++ and may have a size different from that of `int`



# Compatibility of C and C++: One Definition Rule (ODR)

- C allows for multiple tentative definitions of a single global variable in a *single translation unit*, which is disallowed as an *One Definition Rule (ODR)* violation in C++

```
int N;
int N = 10;
```

```
$ gcc main.c main.cpp
main.cpp:46:5: error: redefinition of 'int N'
 int N = 10;
 ^

main.cpp:45:5: note: 'int N' previously declared here
 int N;
 ^
```

- C allows declaring a new type with the same name as an existing **struct**, **union** or **enum** which is not allowed in C++, as in C **struct**, **union** or **enum** types must be indicated as such whenever the type is referenced whereas in C++ all declarations of such types carry the **typedef** implicitly

```
enum BOOL { FALSE, TRUE };
typedef struct _BOOL { int b; } BOOL;
```

```
$ gcc main.c main.cpp
main.cpp:53:33: error: conflicting declaration 'typedef struct _BOOL BOOL'
 typedef struct _BOOL { int b; } BOOL;
 ^~~~

main.cpp:52:6: note: previous declaration as 'enum BOOL'
 enum BOOL { FALSE, TRUE };
 ^~~~~~
```



# Compatibility of C and C++: void Parameter

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- In C, a function prototype without parameters, for example, `int foo();`, implies that the parameters are unspecified. Therefore, it is legal to call such a function with one or more arguments, like `foo(0)`
- In contrast, in C++ a function prototype without arguments means that the function takes no arguments, and calling such a function with arguments is ill-formed
- **In C, declare a function taking no argument by using `void`, as in `int foo(void);`, which is also valid in C++. Empty function prototypes are a deprecated feature in C99 (as they were in C89)**

```
int foo(); int bar(void);
int main() { foo(0); bar(0); }
$ gcc main.c main.cpp
main.c:42:22: error: too many arguments to function 'bar'
 int main() { foo(0); bar(0); }
 ^
main.c:41:16: note: declared here: int foo(); int bar(void);
 ^
main.cpp:59:19: error: too many arguments to function 'int foo()'
 int main() { foo(0); bar(0); }
 ^
main.cpp:58:5: note: declared here: int foo(); int bar(void);
 ^
main.cpp:59:27: error: too many arguments to function 'int bar()'
 int main() { foo(0); bar(0); }
 ^
main.cpp:58:16: note: declared here: int foo(); int bar(void);
 ^
```



# Compatibility of C and C++: Nested struct

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- In both C and C++, one can define nested **struct** types, but the scope is interpreted differently
  - In C++, a nested **struct** is defined only within the scope / namespace of the outer **struct**
  - In C the inner **struct** is also defined outside the outer **struct**

```
struct Outer {
 int o;
 struct Inner {
 int i;
 };
};

struct Outer O1; // Okay in C and C++

#ifndef __cplusplus
struct Inner I1; // Okay only in C
#endif
```



# Compatibility of C and C++: Variable Length Array (VLA)

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- **Variable Length Arrays (VLA)** is a feature where we can allocate an auto array (on stack) of variable size. C supports variable sized arrays from C99 standard
- But, in C++ standard (till C++11) there was no concept of VLA. According to the C++11 standard, array size is a constant-expression. In C++14 mentions array size as a simple expression (not constant-expression)

```
#ifndef __cplusplus
#include <stdio.h>
// VLA in function prototype
int add(int x, int a[*]);
// int add(int x, int a[]); also works
#else
#include <cstdio>
using namespace std;
// Unspecified size in function prototype
int add(int x, int a[]);
#endif
```

```
int set_and_add(int n) { int vals[n]; // Variable Length Array
printf("%d ", sizeof(vals)); // Runtime sizeof
for (int i = 0; i < n; ++i) vals[i] = i;
return add(n, vals);
// vals is declared as an automatic variable
// its lifetime ends when add() returns
}
int main() { int n = 5;
printf("Result = %d", set_and_add(n));
}
int add(int n, int a[]) { int sum = 0;
for (int i = 0; i < n; ++i) sum += a[i];
return sum;
}
```

- The above code uses VLA (`int vals[n]`) in function `set_and_add`. So any size (bounded by a compiler-specified maximum) can be passed to it
- VLA may lead to possibly non-compile time `sizeof` operator



# Compatibility of C and C++: Flexible Array Member (FAM)

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- The last member of a **C99** structure type with more than one member may be a **Flexible Array Member (FAM)**, which takes the syntactic form of an array with unspecified length. This serves a purpose similar to variable-length arrays
- VLAs cannot appear in type definitions, but has defined size (at runtime)
- FAMs have no defined size, but can appear in type definitions
- ISO C++ has no such feature**
- Here is an example of a FAM

```
struct vectord {
 short len; // there must be at least one other data member
 double arr[]; // the flexible array member must be last
 // The compiler may reserve extra padding space here,
 // like it can between struct members
};
```

- Typically, such structures serve as the header in a larger, variable memory allocation

```
struct vectord *vector = malloc(...);
vector->len = ...;
for (int i = 0; i < vector->len; i++)
 vector->arr[i] = ...; // transparently uses the right type (double)
```



# Compatibility of C and C++: restrict

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- `restrict` keyword is mainly used in pointer declarations as a type qualifier for pointers
- It adds no functionality - only informs the compiler about an optimization
- When we use `restrict` with a pointer `ptr`, it tells the compiler that `ptr is the only way to access the object pointed by it`, in other words, `there is no other pointer pointing to the same object`. That is, `restrict` keyword specifies that a particular pointer argument does not alias any other and the compiler does not need to add any additional checks
- If a programmer uses `restrict` keyword and violate the above condition, the behavior is undefined
- `restrict` is supported from C99. **It not supported by ISO C++**

```
#include <stdio.h>
// The purpose of restrict is to show only syntax. It does not change anything in output (or logic)
// It is just a way for programmer to tell compiler about an optimization
void use(int* a, int* b, int* restrict c) {
 *a += *c;
 // Since c is restrict, compiler will not reload value at address c in its assembly code
 // Therefore generated assembly code is optimized
 *b += *c;
}
int main(void) { int a = 50, b = 60, c = 70;
 use(&a, &b, &c);
 printf("%d %d %d", a, b, c);
}
```

Source: [restrict keyword in C](#) and [How to Use the restrict Qualifier in C](#) Accessed 15-Sep-21



# Tutorial Summary

Tutorial T11

Partha Pratim  
Das

Objectives &  
Outline

Why  
Compatibility?

Compatibility of  
C and C++

void\*

const

enum

ODR

void Param

Nested struct

VLA

FAM

restrict

Tutorial Summary

- We have understood why C and C++ incompatible across dialects in spite of C++ being an intended super-set of C
- We studied specific incompatibilities over nearly two dozen features
- We discussed some workarounds to write more compatible code between C and C++



Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

# Programming in Modern C++

Tutorial T12: Compatibility of C and C++: Part 2: Summary

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 Recap

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

- We have understood why C and C++ incompatible across dialects in spite of C++ being an intended super-set of C
- We studied specific incompatibilities over nearly two dozen features
- We discussed some workarounds to write more compatible code between C and C++



# Tutorial Objectives

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

- We present a summary of differences between C and C++

NPTEL



# Tutorial Outline

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

1 Tutorial Recap

2 Summary of Compatibility

3 Tutorial Summary

NPTEL



# Summary of Compatibility

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

NPTEL

## Summary of Compatibility

*We summarize the incompatibility in features already discussed, and also introduce a few new ones in brief*



# Compatibility of C and C++: Summary

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

| C Feature                                                                                                                                                                                                                                                               | C++ Feature                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>Implicit Conversion of <code>void*</code> is allowed in C</li></ul>                                                                                                                                                               | <ul style="list-style-type: none"><li>Implicit conversion is not allowed in C++; allowed only with explicit cast</li></ul>                                                                                               |
| <ul style="list-style-type: none"><li>Implicit Discard of <code>const</code> qualifier for pointer is allowed in C</li></ul>                                                                                                                                            | <ul style="list-style-type: none"><li>Implicit discard is not allowed in C++; allowed only with explicit cast</li></ul>                                                                                                  |
| <ul style="list-style-type: none"><li>Initialization of <code>const</code> Variable is optional in C</li><li>C Standard Library functions have unique signature. For example, <code>strchr</code> in <code>string.h</code></li></ul>                                    | <ul style="list-style-type: none"><li>Initialization is mandatory in C++</li><li>In C++, they may have additional overloaded functions. For example, <code>strchr</code> in <code>cstring</code></li></ul>               |
| <ul style="list-style-type: none"><li>Implicit Conversion of <code>int</code> to <code>enum</code> is allowed in C</li><li><code>enum</code> Enumerators are of type <code>int</code> in C</li></ul>                                                                    | <ul style="list-style-type: none"><li>Implicit conversion is not allowed in C++; allowed only with explicit cast</li><li>Enumerators are of distinct types in C++, having different size from <code>int</code></li></ul> |
| <ul style="list-style-type: none"><li>Multiple definitions of a global in a single translation unit is allowed in C</li><li>Declaring a new type having same name as an existing <code>struct</code>, <code>union</code> or <code>enum</code> is allowed in C</li></ul> | <ul style="list-style-type: none"><li>It is disallowed in C++ due to One Definition Rule (ODR)</li><li>It is disallowed in C++ as all declarations of such types carry the <code>typedef</code> implicitly</li></ul>     |
| <ul style="list-style-type: none"><li>In C, a function prototype without parameters implies that the parameters are unspecified, and can be called with zero or more parameters</li></ul>                                                                               | <ul style="list-style-type: none"><li>In C++, it means zero parameter only</li></ul>                                                                                                                                     |

For compatibility, use `void` for parameter when there is no parameter



# Compatibility of C and C++: Summary

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

| C Feature                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | C++ Feature                                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>Character literals like 'a' are of type <code>int</code> in C. Hence:<ol style="list-style-type: none"><li><code>sizeof('a') = sizeof(int)</code></li><li>'a' is always a signed expression, regardless of whether or not <code>char</code> is a signed or unsigned type</li></ol></li></ul>                                                                                                                                                                                                                                                                                                                            | <ul style="list-style-type: none"><li>They are of type <code>char</code> in C++. Hence:<ol style="list-style-type: none"><li><code>sizeof('a') = sizeof(char) = 1</code></li><li>If 'a' a signed expression or not depends on whether <code>char</code> is a signed or unsigned type, which is implementation specific</li></ol></li></ul> |
| <ul style="list-style-type: none"><li>Boolean type <code>bool</code> is supported in C99 with constants <code>true</code> and <code>false</code>. In C99, a new keyword, <code>_Bool</code>, is introduced as the new Boolean type. The header <code>stdbool.h</code> provides macros <code>bool</code>, <code>true</code> and <code>false</code> that are defined as <code>_Bool</code>, <code>1</code> and <code>0</code>, respectively. Therefore, <code>true</code> and <code>false</code> have type <code>int</code> in C</li><li>For Nested structs, the inner <code>struct</code> is also defined outside the outer <code>struct</code> in C</li></ul> | <ul style="list-style-type: none"><li>In C++, <code>bool</code> is a built-in type with constants <code>true</code> and <code>false</code>. All these are reserved keywords. Conversions to <code>bool</code> are similar to C</li></ul>                                                                                                   |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | A nested <code>struct</code> is defined only within the scope / namespace of the outer <code>struct</code> in C++                                                                                                                                                                                                                          |



# Compatibility of C and C++: Summary

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

| C Feature                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | C++ Feature                                                                                                                                                                                                                                                                                                                                                                    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• <b>inline Function</b> is supported in <b>C99</b>. It is a directive that suggests (but does not require) that the compiler substitute the body of the function inline by inline expansion (saving the overhead of a function call). But it complicates the linkage behavior:<br/><pre>#include &lt;stdio.h&gt; inline int foo() { return 2; } /* Inline in C */ int main() { int ret; /* Driver code */     ret = foo(); /* inline function call */     printf("Output is: %d", ret); }</pre>It gives a linker error <b>undefined reference to 'foo'</b> - as GCC <b>inlines</b>, there is no function call present (<b>foo</b>) inside <b>main</b>. Hence, we fix as:<br/><pre>#include &lt;stdio.h&gt; static /* Inline bound to the this file - no extern linkage */ inline int foo() { return 2; } /* Inline in C */ int main() { int ret; /* Driver code */     ret = foo(); /* inline function call */     printf("Output is: %d", ret); }</pre></li></ul> | <ul style="list-style-type: none"><li>• In C++, the external linkage issues of <b>inline</b> functions are handled by the compiler:<br/><pre>#include &lt;cstdio&gt; using namespace std; inline int foo() {     return 2; } // Inline in C++ int main() {     // Driver code     int ret;     ret = foo(); // inline call     printf("Output is: %d", ret); }</pre></li></ul> |

Source: [inline function specifier](#) Accessed 14-Sep-21



# Compatibility of C and C++: Summary

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

| C Feature                                                                                                                                                                                                                                                                            | C++ Feature                                                                                                                                                                                                                              |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>Variable Length Array (VLA) is supported from C99</li></ul>                                                                                                                                                                                    | <ul style="list-style-type: none"><li>Supported if array size is a constant-expression in C++11 standard and simple expression (not constant-expression) in C++14 standard</li></ul>                                                     |
| <ul style="list-style-type: none"><li>Flexible Array Member (FAM) is supported from C99</li></ul>                                                                                                                                                                                    | <ul style="list-style-type: none"><li>Not supported in ISO C++</li></ul>                                                                                                                                                                 |
| <ul style="list-style-type: none"><li>restrict type qualifier for pointer declarations is supported from C99</li></ul>                                                                                                                                                               | <ul style="list-style-type: none"><li>Not supported in ISO C++, but compilers like GCC, Visual C++, and Intel C++ provide similar functionality as an extension</li></ul>                                                                |
| <ul style="list-style-type: none"><li>Complex arithmetic using the float complex and double complex primitive data types was added in the C99 standard, via the _Complex keyword and complex convenience macro</li></ul>                                                             | <ul style="list-style-type: none"><li>In C++, complex arithmetic can be performed using the complex number class, <i>but the two methods are not code-compatible</i>. (The standards since C++11 require binary compatibility)</li></ul> |
| <ul style="list-style-type: none"><li>Array parameter qualifiers in functions is supported from C89:<br/><code>int foo(int a[const]);</code><br/><i>// equivalent to int *const a</i><br/><code>int bar(char s[static 5]);</code><br/><i>// s is at least 5 chars long</i></li></ul> | <ul style="list-style-type: none"><li>Not supported in ISO C++</li></ul>                                                                                                                                                                 |



# Compatibility of C and C++: Summary

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

| C Feature                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | C++ Feature                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>From C89, <b>Compound Literals</b> is generalized to both built-in and user-defined types by the list initialization syntax of C++11, <i>although with some syntactic and semantic differences</i>:</li> </ul> <pre>struct X { int p, q; }; /* Equivalent in C++ would be X{4, 6} */ struct X a = (struct X){4, 6};</pre>                                                                                                                                                                                                | <ul style="list-style-type: none"> <li>The following works in C++11 onward:</li> </ul> <pre>struct X { int p, q; }; struct X a = (struct X){4, 6}; struct X b = X{4, 6};</pre>                                                                                                                                                                                                                                                          |
| <ul style="list-style-type: none"> <li>From C89, <b>Designated Initializers</b> for structs and arrays are allowed in C. Designated initializers allow members to be initialized by name, in any order, and without explicitly providing the preceding values:</li> </ul> <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name =     { .z ="Pi", .x =3, .y =3.1415 }; /* Only C */ char s[20] = {[0] ='a', [8] ='g'}; /* Only C */  #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre> | <ul style="list-style-type: none"> <li>These are not allowed in C++. struct designated initializers are planned for addition in C++2x</li> </ul> <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name =     { .z ="Pi", .x =3, .y =3.1415 }; // C++2x ?  char s[20] = {[0] ='a', [8] ='g'}; // No C++ Plan  #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre> |



# Compatibility of C and C++: Summary

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

| C Feature                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | C++ Feature                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li><b>_Noreturn function specifier</b> marks function in C that does not return with a value or implicitly after completion. It may <b>return by executing longjmp</b>:</li></ul> <pre>#include &lt;stdio.h&gt;  /* Nothing to return */ _Noreturn void show() { printf("BYE BYE"); } int main() {     printf("Ready to begin...");     show();      printf("NOT over till now"); }</pre> <p>Compiler Warning:<br/>'noreturn' function does return</p> <p>Output is:<br/>Ready to begin...BYE BYE</p> <p>Source: <a href="#">_Noreturn function specifier</a> Accessed 14-Sep-21</p> | <ul style="list-style-type: none"><li><b>[[noreturn]] attribute</b> marks function in C++ that does not return with a value or implicitly after completion. It may <b>throw</b>:</li></ul> <pre>#include &lt;cstdio&gt; using namespace std;  /* Nothing to return */ [[noreturn]] void show() { printf("BYE BYE"); } int main() {     printf("Ready to begin...");     show();      printf("NOT over till now"); }</pre> <p>Compiler Warning:<br/>'noreturn' function does return</p> <p>Output:<br/>Ready to begin...BYE BYE</p> <p>Source: <a href="#">C++ attribute: noreturn</a> Accessed 14-Sep-21</p> |

*This must not be confused with void return type used for functions that return, but without a value*



# Compatibility of C and C++: Summary

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

| C Feature                                                                                                                                                                                                                                                                                                                                                                                                               | C++ Feature                                                                                                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• <b>Extra C++ reserved words</b> may fail C codes in C++ compiler. The following code works fine in C.</li></ul> <pre>struct template {<br/>    int new;<br/>    struct template* class;<br/>};</pre>                                                                                                                                                                            | <ul style="list-style-type: none"><li>• It naturally fails in C++:</li></ul> <pre>struct template { // template is a reserved word<br/>    int new; // new is a reserved word<br/>    struct template*<br/>        class; // class is a reserved word<br/>};</pre>                                                           |
| <ul style="list-style-type: none"><li>• We can observe <b>several mixed effects</b> in C and C++ due to incompatibility where the code compiles in both languages but behaves differently</li></ul> <pre>#include &lt;stdio.h&gt;<br/>extern int T;<br/>int size(void) { struct T { int i; int j; };<br/>    return sizeof(T);<br/>    /* C: return sizeof(int) */<br/>}<br/>int main() { printf("%d", size()); }</pre> | <ul style="list-style-type: none"><li>• It naturally fails in C++:</li></ul> <pre>#include &lt;cstdio&gt;<br/>using namespace std;<br/>extern int T;<br/>int size(void) { struct T { int i; int j; };<br/>    return sizeof(T);<br/>    // C++: return sizeof(struct T)<br/>}<br/>int main() { printf("%d", size()); }</pre> |



# Tutorial Summary

Tutorial T12

Partha Pratim  
Das

Tutorial Recap

Objectives &  
Outline

Summary of  
Compatibility

Tutorial Summary

- We presented a summary of differences table between **C99** and **C++11**

NPTEL