Programming in Modern C++

Module M06: Constants and Inline Functions

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

Weekly Recap

- Understood the importance and ease of C++ in programming
- KYC - Pre-requisites, Outline, Evaluation and Textbooks and References
- Understood some fundamental differences between C & C++:
  - IO, Variable declaration, and Loops
  - Arrays and Strings
  - Sorting and Searching
  - Stack and Common Containers in C++
  - Various Standard Library in C and in C++

Module Objectives

- Understand `const` in C++ and contrast with *Manifest Constants*
- Understand `inline` in C++ and contrast with *Macros*

Module Outline

1. Weekly Recap
2. cv-qualifier: const & volatile
   - Notion of const
   - Advantages of const
   - const and pointer
     - C-String
   - Notion of volatile
3. inline functions
   - Macros with Params in C
     - Pitfalls of Macros
   - Notion of inline
     - Comparison of Macros and inline Functions
     - Limitations of inline Functions
4. Module Summary

const-ness and cv-qualifier

Program 06.01: Manifest constants in C / C++

- Manifest constants are defined by #define
- Manifest constants are replaced by CPP (C Pre-Processor). Check Tutorial on **C Preprocessor (CPP)**

Source Program

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

#define TWO 2            // Manifest const
#define PI 4.0*atan(1.0) // Const expr.

int main() { int r = 10;
    double peri = TWO * PI * r;
    cout << "Perimeter = " << peri << endl;
}
```

```
Perimeter = 62.8319
```

- TWO is a manifest constant
- PI is a manifest constant as macro
- TWO & PI look like variables

Program after CPP

```
// Contents of <iostream> header replaced by CPP
// Contents of <cmath> header replaced by CPP
using namespace std;

// #define of TWO consumed by CPP
// #define of PI consumed by CPP

int main() { int r = 10;
    double peri = 2 * 4.0*atan(1.0) * r; // By CPP
    cout << "Perimeter = " << peri << endl;
}
```

```
Perimeter = 62.8319
```

- CPP replaces the token TWO by 2
- CPP replaces the token PI by 4.0*atan(1.0) and evaluates
- Compiler *sees* them as constants
- TWO * PI = 6.28319 by constant folding of compiler

Notion of const-ness

- The value of a const variable *cannot be changed after definition*

```
const int n = 10; // n is an int type variable with value 10. n is a constant
...
n = 5; // Is a compilation error as n cannot be changed
...
int m;
int *p = 0;
p = &m; // Hold m by pointer p
*p = 7; // Change m by p; m is now 7
...
p = &n; // Is a compilation error as n may be changed by *p = 5;
```

- Naturally, a const variable *must be initialized when defined*

```
const int n; // Is a compilation error as n must be initialized
```

- A variable of *any data type* can be declared as const

```
typedef struct _Complex {
    double re;
    double im;
} Complex;
const Complex c = {2.3, 7.5}; // c is a Complex type variable
                              // It is initialized with c.re = 2.3 and c.im = 7.5. c is a constant
...
c.re = 3.5; // Is a compilation error as no part of c can be changed
```

Program 06.02: Compare #define and const

Using #define

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

#define TWO 2
#define PI 4.0*atan(1.0)

int main() { int r = 10;
    // Replace by CPP
    double peri = 2 * 4.0*atan(1.0) * r;
    cout << "Perimeter = " << peri << endl;
}
```

```
Perimeter = 62.8319
```

- TWO is a manifest constant
- PI is a manifest constant
- TWO & PI *look like variables*
- Types of TWO & PI may be indeterminate
- TWO * PI = 6.28319 by constant folding of compiler

Using const

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

const int TWO = 2;
const double PI = 4.0*atan(1.0);

int main() { int r = 10;
    // No replacement by CPP
    double peri = TWO * PI * r;
    cout << "Perimeter = " << peri << endl;
}
```

```
Perimeter = 62.8319
```

- TWO is a const variable initialized to 2
- PI is a const variable initialized to 4.0*atan(1.0)
- TWO & PI *are variables*
- Type of TWO is const int
- Type of PI is const double

Advantages of const

- Natural Constants like π, e, Φ (*Golden Ratio*) etc. can be compactly defined and used

```
const double pi = 4.0*atan(1.0);           // pi = 3.14159
const double e = exp(1.0);                 // e = 2.71828
const double phi = (sqrt(5.0) + 1) / 2.0;  // phi = 1.61803

const int TRUE = 1;                        // Truth values
const int FALSE = 0;

const int null = 0;                        // null value
```

Note: NULL is a manifest constant in C/C++ set to 0

- Program Constants like number of elements, array size etc. can be defined at one place (at times in a header) and used all over the program

```
const int nArraySize = 100;
const int nElements = 10;

int main() {
    int A[nArraySize];                    // Array size
    for (int i = 0; i < nElements; ++i)   // Number of elements
        A[i] = i * i;
}
```

Advantages of const

- Prefer const over #define

Using #define	**Using** const
Manifest Constant	**Constant Variable**
• Is not type safe • Replaced textually by CPP • Cannot be *watched* in debugger • Evaluated as *many times as replaced*	• Has its type • Visible to the compiler • Can be *watched* in debugger • Evaluated *only on initialization*

const and Pointers

- const-ness can be used with Pointers in one of the two ways:
  - **Pointer to Constant data** where the *pointee* (pointed data) cannot be changed
  - **Constant Pointer** where the *pointer* (address) cannot be changed
- Consider usual pointer-pointee computation (without const):

```
int m = 4;
int n = 5;
int * p = &n; // p points to n. *p is 5
...
n = 6;        // n and *p are 6 now
*p = 7;       // n and *p are 7 now. POINTEE changes
...
p = &m;       // p points to m. *p is 4. POINTER changes
*p = 8;       // m and *p are 8 now. n is 7. POINTEE changes
```

const and Pointers: *Pointer to Constant data*

Consider pointed data

```
int m = 4;
const int n = 5;
const int * p = &n;
...
n = 6;  // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
p = &m; // Okay
*p = 8; // Error: p points to a constant data. Its pointee cannot be changed
```

Interestingly,

```
int n = 5;
const int * p = &n;
...
n = 6;  // Okay
*p = 6; // Error: p points to a constant data (n) that cannot be changed
```

Finally,

```
const int n = 5;
int *p = &n; // Error: If this were allowed, we would be able to change constant n
...
n = 6;  // Error: n is constant and cannot be changed
*p = 6; // Would have been okay, if declaration of p were valid
```

const and Pointers: *Constant Pointer*

Consider pointer

```
int m = 4, n = 5;
int * const p = &n;
...
n = 6;  // Okay
*p = 7; // Okay. Both n and *p are 7 now
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

By extension, both can be const

```
const int m = 4;
const int n = 5;
const int * const p = &n;
...
n = 6;  // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Finally, to decide on const-ness, draw a mental line through *

```
int n = 5;
int * p = &n;             // non-const-Pointer to non-const-Pointee
const int * p = &n;       // non-const-Pointer to const-Pointee
int * const p = &n;       // const-Pointer to non-const-Pointee
const int * const p = &n; // const-Pointer to const-Pointee
```

const and Pointers: The case of C-string

Consider the example:

```
char * str = strdup("IIT, Kharagpur");
str[0] = ’N’;                     // Edit the name
cout << str << endl;
str = strdup("JIT, Kharagpur"); // Change the name
cout << str << endl;
```

Output is:

```
NIT, Kharagpur
JIT, Kharagpur
```

To stop editing the name:

```
const char * str = strdup("IIT, Kharagpur");
str[0] = ’N’;                     // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Change the name
```

To stop changing the name:

```
char * const str = strdup("IIT, Kharagpur");
str[0] = ’N’;                     // Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

To stop both:

```
const char * const str = strdup("IIT, Kharagpur");
str[0] = ’N’;                     // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

Notion of volatile

- Variable Read-Write
  - The value of a variable can be *read and / or assigned* at any point of time
  - The value assigned to a variable does not change till a next assignment is made (*value is persistent*)
- `const`
  - The value of a `const` variable can be set *only at initialization* – *cannot be changed* afterwards
- `volatile`
  - In contrast, the value of a `volatile` variable may be different every time it is read – *even if no assignment has been made to it*
  - A variable is taken as `volatile` if it can be *changed by hardware, the kernel, another thread* etc.
- `cv-qualifier`: A declaration may be prefixed with a qualifier – `const` or `volatile`

Using volatile

Consider:

```
static int i;
void fun(void) {
    i = 0;
    while (i != 100);
}
```

This is an *infinite loop*! Hence the compiler should optimize as:

```
static int i;
void fun(void) {
    i = 0;
    while (1);          // Compiler optimizes
}
```

Now qualify i as volatile:

```
static volatile int i;
void fun(void) {
    i = 0;
    while (i != 100);  // Compiler does not optimize
}
```

Being volatile, i can be changed by hardware anytime. *It waits till the value becomes 100* (possibly some hardware writes to a port).

inline functions

inline **functions**

Program 06.03: Macros with Parameters

- Macros with Parameters are defined by #define
- Macros with Parameters are replaced by CPP

Source Program

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

#define SQUARE(x) x * x

int main() {
    int a = 3, b;

    b = SQUARE(a);

    cout << "Square = " << b << endl;
}
```

Square = 9

- SQUARE(x) is a macro with one param
- SQUARE(x) looks like a function

Program after CPP

```
#include <iostream> // Header replaced by CPP
using namespace std;

// #define of SQUARE(x) consumed by CPP

int main() {
    int a = 3, b;

    b = a * a; // Replaced by CPP

    cout << "Square = " << b << endl;
}
```

Square = 9

- CPP replaces the SQUARE(x) substituting x with a
- Compiler does not *see* it as function

Pitfalls of macros

Consider the example:

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

#define SQUARE(x) x * x

int main() {
    int a = 3, b;

    b = SQUARE(a + 1); // Error: Wrong macro expansion

    cout << "Square = " << b << endl;
}
```

Output is 7 in stead of 16 as expected. On the expansion line it gets:

```
b = a + 1 * a + 1;
```

To fix:

```
#define SQUARE(x) (x) * (x)
```

Now:

```
b = (a + 1) * (a + 1);
```

Pitfalls of macros

Continuing ...

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

#define SQUARE(x) (x) * (x)

int main() {
    int a = 3, b;

    b = SQUARE(++a);

    cout << "Square = " << b << endl;
}
```

Output is 25 in stead of 16 as expected. On the expansion line it gets:

```
b = (++a) * (++a);
```

and a is *incremented twice* before being used! There is no easy fix.

inline Function

- An inline function is just a function like any other
- The function prototype is preceded by the keyword inline
- An inline function is *expanded* (*inlined*) at the site of its call and the overhead of passing parameters between caller and callee (or called) functions is avoided

Program 06.04: Macros as inline Functions

- Define the function
- Prefix function header with inline
- *Compile function body and function call together*

Using macro

```
#include <iostream>
using namespace std;
#define SQUARE(x) x * x
int main() {
    int a = 3, b;
    b = SQUARE(a);
    cout << "Square = " << b << endl;
}
```

Square = 9

- SQUARE(x) is a macro with one param
- Macro SQUARE(x) is efficient
- SQUARE(a + 1) fails
- SQUARE(++a) fails
- SQUARE(++a) does not check type

Using inline

```
#include <iostream>
using namespace std;
inline int SQUARE(int x) { return x * x; }
int main() {
    int a = 3, b;
    b = SQUARE(a);
    cout << "Square = " << b << endl;
}
```

Square = 9

- SQUARE(x) is a function with one param
- inline SQUARE(x) is equally efficient
- SQUARE(a + 1) works
- SQUARE(++a) works
- SQUARE(++a) checks type

Macros & inline Functions: Compare and Contrast

Macros	inline **Functions**
Expanded at the place of calls	Expanded at the place of calls
Efficient in execution	Efficient in execution
Code bloats	Code bloats
Has *syntactic and semantic pitfalls*	*No pitfall*
Type checking for parameters is *not done*	*Type checking* for parameters is *robust*
Helps to write `max` / `swap` for *all types*	Needs `template` to support *all types*
Errors are *not checked during compilation*	Errors are *checked during compilation*
Not available to debugger	*Available* to debugger in DEBUG build

Limitations of Function `inline`ing

- `inline`ing is a *directive* – compiler may not inline functions with large body
- `inline` functions may not be *recursive*
- Function body is needed for `inline`ing at the time of function call. Hence, implementation hiding is not possible. *Implement* `inline` *functions in header files*
- `inline` functions *must not have two different definitions*

Module Summary

- Revisited manifest constants from C
- Understood const-ness, its use and advantages over manifest constants, and its interplay with pointers
- Understood the notion and use of volatile data
- Revisited macros with parameters from C
- Understood inline functions, their advantages over macros, and their limitations

Programming in Modern C++

Module M07: Reference & Pointer

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

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

Module Recap

- Revisited manifest constants from C
- Understood `const`-ness, its use and advantages over manifest constants, and its interplay with pointers
- Understood the notion and use of `volatile` data
- Revisited macros with parameters from C
- Understood `inline` functions, their advantages over macros, and their limitations

Module Objectives

- Understand References in C++
- Compare and contrast References and Pointers

Module Outline

1. Reference Variable
   - Pitfalls in Reference
2. Call-by-Reference
   - Simple C Program to swap
   - Simple C/C++ Program to swap two numbers
   - const Reference Parameter
3. Return-by-Reference
   - Pitfalls of Return-by Reference
4. I/O Parameters of a Function
5. Recommended Call and Return Mechanisms
6. Difference between Reference and Pointer
7. Module Summary

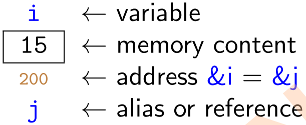
Reference Variable

Reference Variable

Reference

- A reference is an alias / synonym for an existing variable

```
int i = 15;   // i is a variable
int &j = i;   // j is a reference to i
```



i ← variable

$\boxed{15}$ ← memory content

200 ← address &i = &j

j ← alias or reference

Program 07.01: Behavior of Reference

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

int main() {
    int a = 10, &b = a; // b is reference of a

    // a and b have the same memory location
    cout << "a = " << a << ", b = " << b << ". " << "&a = " << &a << ", &b = " << &b << endl;

    ++a; // Changing a appears as change in b
    cout << "a = " << a << ", b = " << b << endl;

    ++b; // Changing b also changes a
    cout << "a = " << a << ", b = " << b << endl;
}
```

```
a = 10, b = 10. &a = 002BF944, &b = 002BF944
a = 11, b = 11
a = 12, b = 12
```

- a and b have the *same memory location* and hence *the same value*
- Changing one changes the other and vice-versa

Pitfalls in Reference

Wrong declaration	Reason	Correct declaration
`int& i;`	no variable (address) to refer to – must be initialized	`int& i = j;`
`int& j = 5;`	no address to refer to as 5 is a *constant*	`const int& j = 5;`
`int& i = j + k;`	only temporary address (result of `j + k`) to refer to	`const int& i = j + k;`

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

int main() {
    int i = 2;
    int& j = i;
    const int& k = 5;     // const tells compiler to allocate a memory with the value 5
    const int& l = j + k; // Similarly for j + k = 7 for l to refer to

    cout << i << ", " << &i << endl;    // Prints: 2, 0x61fef8
    cout << j << ", " << &j << endl;    // Prints: 2, 0x61fef8
    cout << k << ", " << &k << endl;    // Prints: 5, 0x61fefc
    cout << l << ", " << &l << endl;    // Prints: 7, 0x61ff00
}
```

Call-by-Reference

C++ Program 07.02: Call-by-Reference

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

void Function_under_param_test( // Function prototype
    int&, // Reference parameter
    int); // Value parameter

int main() { int a = 20;
    cout << "a = " << a << ", &a = " << &a << endl << endl;
    Function_under_param_test(a, a); // Function call
}
void Function_under_param_test(int &b, int c) {  // Function definition
    cout << "b = " << b << ", &b = " << &b << endl << endl;
    cout << "c = " << c << ", &c = " << &c << endl << endl;
}
------- Output -------
a = 20, &a = 0023FA30
b = 20, &b = 0023FA30   // Address of b is same as a as b is a reference of a
c = 20, &c = 0023F95C   // Address different from a as c is a copy of a
```

- Param b is *call-by-reference* while param c is *call-by-value*
- Actual param a and formal param b get the *same value* in called function
- Actual param a and formal param c get the *same value* in called function
- Actual param a and formal param b get the *same address* in called function
- However, actual param a and formal param c have *different addresses* in called function

C Program 07.03: Swap in C

Call-by-value – wrong

```
#include <stdio.h>

void swap(int, int); // Call-by-value
int main() { int a = 10, b = 15;
    printf("a= %d & b= %d to swap\n", a, b);
    swap(a, b);
    printf("a= %d & b= %d on swap\n", a, b);
}
void swap(int c, int d) { int t;
    t = c; c = d; d = t;
}
```

- a= 10 & b= 15 to swap
- a= 10 & b= 15 on swap // No swap

- Passing values of a=10 & b=15
- In callee; c = 10 & d = 15
- Swapping the values of c & d
- No change for the values of a & b in caller
- Swapping the value of c & d instead of a & b

Call-by-address – right

```
#include <stdio.h>

void swap(int *, int *); // Call-by-address
int main() { int a=10, b=15;
    printf("a= %d & b= %d to swap\n", a, b);
    swap(&a, &b);  // Unnatural call
    printf("a= %d & b= %d on swap\n", a, b);
}
void swap(int *x, int *y) { int t;
    t = *x; *x = *y; *y = t;
}
```

- a= 10 & b= 15 to swap
- a= 15 & b= 10 on swap // Correct swap

- Passing Address of a & b
- In callee x = Addr(a) & y = Addr(b)
- Values at the addresses is swapped
- Desired changes for the values of a & b in caller
- It is correct, but C++ has a better way out

Program 07.04: Swap in C & C++

C Program: Call-by-value – wrong

```
#include <stdio.h>

void swap(int, int); // Call-by-value
int main() { int a = 10, b = 15;
    printf("a= %d & b= %d to swap\n",a,b);
    swap(a, b);
    printf("a= %d & b= %d on swap\n",a,b);
}
void swap(int c, int d) { int t ;
    t = c; c = d; d = t;
}
```

- a= 10 & b= 15 to swap
- a= 10 & b= 15 on swap // No swap

- Passing values of a=10 & b=15
- In callee; c = 10 & d = 15
- Swapping the values of c & d
- No change for the values of a & b in caller
- Here c & d do not share address with a & b

C++ Program: Call-by-reference – right

```
#include <iostream>
using namespace std;
void swap(int&, int&); // Call-by-reference
int main() { int a = 10, b = 15;
    cout<<"a= "<<a<<" & b= "<<b<<"to swap"<<endl;
    swap(a, b);  // Natural call
    cout<<"a= "<<a<<" & b= "<<b<<"on swap"<<endl;
}
void swap(int &x, int &y) { int t ;
    t = x; x = y; y = t;
}
```

- a= 10 & b= 15 to swap
- a= 15 & b= 10 on swap // Correct swap

- Passing values of a = 10 & b = 15
- In callee: x = 10 & y = 15
- Swapping the values of x & y
- Desired changes for the values of a & b in caller
- x & y having *same address* as a & b respectively

Program 07.05: Reference Parameter as const

- A reference parameter may get changed in the called function
- Use const to stop reference parameter being changed

const **reference** – **bad**

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

int Ref_const(const int &x) {
    ++x;          // Not allowed
    return (x);
}
int main() { int a = 10, b;
    b = Ref_const(a);
    cout << "a = " << a <<" and"
         << " b = " << b;
}
```

- Error: Increment of read only Reference 'x'

- Compilation Error: Value of x cannot be changed
- Implies, a cannot be changed through x

const **reference** – **good**

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

int Ref_const(const int &x) {

    return (x + 1);
}
int main() { int a = 10, b;
    b = Ref_const(a);
    cout << "a = " << a << " and"
         << " b = " << b;
}
```

a = 10 and b = 11

- No violation

Return-by-Reference

Program 07.06: Return-by-Reference

- A function can return a value by reference (Return-by-Reference)
- C uses Return-by-value

Return-by-value

```
#include <iostream>
using namespace std;
int Function_Return_By_Val(int &x) {
    cout << "x = " << x << " &x = " << &x << endl;
    return (x);
}
int main() { int a = 10;
    cout << "a = " << a << " &a = " << &a << endl;
    const int& b = // const needed. Why?
        Function_Return_By_Val(a);
    cout << "b = " << b << " &b = " << &b << endl;
}
```

```
a = 10 &a = 00DCFD18
x = 10 &x = 00DCFD18
b = 10 &b = 00DCFD00 // Reference to temporary
```

- Returned variable is *temporary*
- Has a *different address*

Return-by-reference

```
#include <iostream>
using namespace std;
int& Function_Return_By_Ref(int &x) {
    cout << "x = " << x << " &x = " << &x << endl;
    return (x);
}
int main() { int a = 10;
    cout << "a = " << a << " &a = " << &a << endl;
    const int& b = // const optional
        Function_Return_By_Ref(a);
    cout << "b = " << b << " &b = " << &b << endl;
}
```

```
a = 10 &a = 00A7F8FC
x = 10 &x = 00A7F8FC
b = 10 &b = 00A7F8FC // Reference to a
```

- Returned variable is *an alias of* a
- Has the *same address*

Program 07.07: Return-by-Reference can get tricky

Return-by-reference

```
#include <iostream>
using namespace std;
int& Return_ref(int &x) {


    return (x);
}
int main() { int a = 10, b = Return_ref(a);
    cout << "a = " << a << " and b = "
         << b << endl;

    Return_ref(a) = 3; // Changes variable a
    cout << "a = " << a;
}
```

```
a = 10 and b = 10
a = 3
```

- Note how *a value is assigned to function call*
- This can change a local variable

Return-by-reference – Risky!

```
#include <iostream>
using namespace std;
int& Return_ref(int &x) {
    int t = x;
    t++;
    return (t);
}
int main() { int a = 10, b = Return_ref(a);
    cout << "a = " << a << " and b = "
         << b << endl;

    Return_ref(a) = 3; // Changes local t
    cout << "a = " << a;
}
```

```
a = 10 and b = 11
a = 10
```

- We expect a to be 3, *but it has not changed*
- It returns reference to local. This is *risky*

I/O Parameters of a Function

I/O Parameters of a Function

I/O of a Function

- In C++ we can change values with a function as follows:

I/O of Function	Purpose	Mechanism
Value Parameter	Input	Call-by-value
Reference Parameter	In-Out	Call-by-reference
const Reference Parameter	Input	Call-by-reference
Return Value	Output	Return-by-value Return-by-reference const Return-by-reference

- In addition, we can use the Call-by-address (Call-by-value with pointer) and Return-by-address (Return-by-value with pointer) as in C
- But it is neither required nor advised

Recommended Mechanisms

Recommended Mechanisms

Recommended Mechanisms

- **Call**
  - Pass parameters of built-in types *by* **value**
    - ▷ Recall: *Array parameters* are passed *by* **reference in C and C++**
  - Pass parameters of user-defined types *by* **reference**
    - ▷ Make a *reference parameter* `const` if it is not used for output
- **Return**
  - Return built-in types *by* **value**
  - Return user-defined types *by* **reference**
    - ▷ Return value *is not copied back*
    - ▷ May be *faster* than returning a value
    - ▷ Beware: Calling function *can change returned object*
    - ▷ Never return a local variables by reference

Difference between Reference and Pointer

Difference between Reference and Pointer

Difference between Reference and Pointer

Pointers	References
• Refers to an *address (exposed)* • Pointers can point to `NULL`	• Refers to an *address (hidden)* • References cannot be `NULL`
int *p = NULL; // p is not pointing	int &j ; // wrong
• Pointers can point to *different variables* at *different times*	• For a reference, its *referent is fixed*

`int a, b, *p;`
`p = &a; // p points to a`
`...`
`p = &b; // p points to b`
	`int a, c, &b = a; // Okay`
	`...`
	`&b = c            // Error`

• `NULL` checking *is required*	• *Does not require* `NULL` checking • Makes code *faster*
• *Allows* users to *operate on the address* • diff pointers, increment, etc.	• *Does not allow* users to *operate on the address* • All operations are interpreted for the referent
• Array of pointers can be *defined*	• Array of references *not allowed*

Module Summary

- Introduced reference in C++
- Studied the difference between call-by-value and call-by-reference
- Studied the difference between return-by-value and return-by-reference
- Discussed the difference between References and Pointers

Programming in Modern C++

Module M08: Default Parameters & Function Overloading

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

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

Module Recap

- Introduced reference in C++
- Studied the difference between call-by-value and call-by-reference
- Studied the difference between return-by-value and return-by-reference
- Discussed the difference between References and Pointers

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Default Parameters in Overloading

Module Objectives

- Understand Default Parameters
- Understand Function Overloading and Resolution

Module Outline

1. Default Parameters
   - Examples
   - Highlights
   - Restrictions on default parameters
2. Function Overloading
   - Examples
   - Restrictions
   - Rules
3. Overload Resolution
   - Exact Match
   - Promotion & Conversion
   - Examples
   - Ambiguity
4. How to overload Default Parameter
5. Module Summary

Default Parameters

Motivation: Example CreateWindow in MSVC++

Declaration of CreateWindow

```
HWND WINAPI CreateWindow(
    _In_opt_ LPCTSTR    lpClassName,
    _In_opt_ LPCTSTR    lpWindowName,
    _In_     DWORD      dwStyle,
    _In_     int        x,
    _In_     int        y,
    _In_     int        nWidth,
    _In_     int        nHeight,
    _In_opt_ HWND       hWndParent,
    _In_opt_ HMENU      hMenu,
    _In_opt_ HINSTANCE hInstance,
    _In_opt_ LPVOID     lpParam
);
```

Calling CreateWindow

```
hWnd = CreateWindow(
    ClsName,
    WndName,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
);
```

- There are 11 parameters in `CreateWindow()`
- Of these 11, 8 parameters (4 are `CWUSEDEFAULT`, 3 are `NULL`, and 1 is `hInstance`) usually get same values in most calls
- Instead of using these 8 fixed valued Parameters at call, we may assign the *values in formal parameter*
- C++ allows us to do so through the mechanism called **Default parameters**

Program 08.01: Function with a default parameter

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

int IdentityFunction(int a = 10) { // Default value for parameter a
    return (a);
}

int main() {
    int x = 5, y;

    y = IdentityFunction(x);  // Usual function call. Actual parameter taken as x = 5
    cout << "y = " << y << endl;

    y = IdentityFunction();   // Uses default parameter. Actual parameter taken as 10
    cout << "y = " << y << endl;
}
----------
y = 5
y = 10
```

Program 08.02: Function with 2 default parameters

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

int Add(int a = 10, int b = 20) {
    return (a + b);
}
int main() { int x = 5, y = 6, z;

    z = Add(x, y); // Usual function call -- a = x = 5 & b = y = 6
    cout << "Sum = " << z << endl;

    z = Add(x);    // One parameter defaulted -- a = x = 5 & b = 20
    cout << "Sum = " << z << endl;

    z = Add();     // Both parameter defaulted -- a = 10 & b = 20
    cout << "Sum = " << z << endl;
}
----------
Sum = 11
Sum = 25
Sum = 30
```

Default Parameter: Highlighted Points

- C++ allows programmer to assign default values to the function parameters
- Default values are specified while prototyping the function
- Default parameters are required while calling functions with fewer arguments or without any argument
- Better to use default value for less used parameters
- Default arguments may be expressions also

Restrictions on default parameters

- *All parameters to the right of a parameter with default argument* **must have** *default arguments* (function f violates)
- *Default arguments* **cannot be** *re-defined* (second signature of function g violates)
- *All non-defaulted parameters* **needed** in a *call* (first call of g() violates)

```
#include <iostream>

void f(int, double = 0.0, char *);
// Error C2548: f: missing default parameter for parameter 3

void g(int, double = 0, char * = NULL); // OK
void g(int, double = 1, char * = NULL);
// Error C2572: g: redefinition of default parameter : parameter 3
// Error C2572: g: redefinition of default parameter : parameter 2

int main() {
    int i = 5; double d = 1.2; char c = 'b';

    g(); // Error C2660: g: function does not take 0 arguments
    g(i);
    g(i, d);
    g(i, d, &c);
}
```

Restrictions on default parameters

- Default parameters to be supplied *only in a header file* and *not in the definition* of a function

```
// Header file: myFunc.h
void g(int, double, char = ’a’); // Defaults ch
void g(int i, double f = 0.0, char ch); // A new overload. Defaults f & ch
void g(int i = 0, double f, char ch);   // A new overload. Defaults i, f & ch
// void g(int i = 0, double f = 0.0, char ch = ’a’); // Alternate signature. Defaults all in one go
----------------------------------------------------
// Source File
#include <iostream>
using namespace std;
#include "myFunc.h" // Defaults taken from header
void g(int i, double d, char c) { cout << i << ’ ’ << d << ’ ’ << c << endl; } // No defaults here
----------------------------------------------------
// Application File
#include <iostream>
#include "myFunc.h"
int main() { int i = 5; double d = 1.2; char c = ’b’;
    g();          // Prints: 0 0 a
    g(i);         // Prints: 5 0 a
    g(i, d);      // Prints: 5 1.2 a
    g(i, d, c);   // Prints: 5 1.2 b
}
```

Function Overloading

Function Overloading

Function overloads: Matrix Multiplication in C

- *Similar functions* with *different* **data types** and **algorithms**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply_M_M  (Mat a,    Mat b,    Mat* c);    // c = a * b
void Multiply_M_VC (Mat a,    VecCol b, VecCol* c); // c = a * b
void Multiply_VR_M (VecRow a, Mat b,    VecRow* c); // c = a * b
void Multiply_VC_VR(VecCol a, VecRow b, Mat* c);    // c = a * b
void Multiply_VR_VC(VecRow a, VecCol b, int* c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply_M_M  (m1, m2, &rm);  // rm  <-- m1 * m2
    Multiply_M_VC (m1, cv, &rcv); // rcv <-- m1 * cv
    Multiply_VR_M (rv, m2, &rrv); // rrv <-- rv * m2
    Multiply_VC_VR(cv, rv, &rm);  // rm  <-- cv * rv
    Multiply_VR_VC(rv, cv, &r);   // r   <-- rv * cv
    return 0;
}
```

- 5 multiplication functions share *similar functionality* but *different argument types*
- C treats them by 5 different function names. Makes it difficult for the user to remember and use
- **C++ has an elegant solution**

Function overloads: Matrix Multiplication in C++

- Functions *having the same* **name**, *similar functionality* but *different* **algorithms**, and identified by *different interfaces* **data types**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply(const Mat& a,    const Mat& b,    Mat& c);    // c = a * b
void Multiply(const Mat& a,    const VecCol& b, VecCol& c); // c = a * b
void Multiply(const VecRow& a, const Mat& b,    VecRow& c); // c = a * b
void Multiply(const VecCol& a, const VecRow& b, Mat& c);    // c = a * b
void Multiply(const VecRow& a, const VecCol& b, int& c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm);  // rm  <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm);  // rm  <-- cv * rv
    Multiply(rv, cv, r);   // r   <-- rv * cv
    return 0;
}
```

- These **5 functions** having *different argument types* are represented as *one function name* (`Multiply`) in C++
- This is called **Function Overloading** or **Static Polymorphism**

Program 08.03/04: Function Overloading

- Define *multiple functions* having the *same* **name**
- *Binding* happens at **compile time**

Same # of Parameters

```
#include <iostream>
using namespace std;
int Add(int a, int b) { return (a + b); }
double Add(double c, double d) { return (c + d); }
int main() {
    int x = 5, y = 6, z;
    z = Add(x, y); // int Add(int, int)
    cout << "int sum = " << z;

    double s = 3.5, t = 4.25, u;
    u = Add(s, t); // double Add(double, double)
    cout << "double sum = " << u << endl;
}
```

```
int sum = 11 double sum = 7.75
```

- Same Add function to add two ints or two doubles
- Same # of parameters but *different types*

Different # of Parameters

```
#include <iostream>
using namespace std;
int Area(int a, int b)  return (a * b);
int Area(int c) { return (c * c); }
int main() {
    int x = 10, y = 12, z = 5, t;
    t = Area(x, y); // int Area(int, int)
    cout << "Area of Rectangle = " << t;

    int z = 5, u;
    u = Area(z); // int Area(int)
    cout << " Area of Square = " << u << endl;
}
```

```
Area of Rectangle = 12 Area of Square = 25
```

- Same Area function for *rectangles* and for squares
- *Different number of parameters*

Program 08.05: Restrictions in Function Overloading

- Two functions having the *same signature* but *different return types* cannot be overloaded

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

int    Area(int a, int b) { return (a * b); }
double Area(int a, int b) { return (a * b); }
// Error C2556: double Area(int,int): overloaded function differs only by return type
//              from int Area(int,int)
// Error C2371: Area: redefinition; different basic types

int main() {
    int x = 10, y = 12, z = 5, t;
    double f;

    t = Area(x, y);
    // Error C2568: =: unable to resolve function overload
    // Error C3861: Area: identifier not found

    cout << "Multiplication = " << t << endl;

    f = Area(y, z); // Errors C2568 and C3861 as above
    cout << "Multiplication = " << f << endl;
}
```

Function Overloading: Summary of Rules

- The *same function name* may be used in *several definitions*
- Functions with the *same name* must have *different number of formal parameters* and/or *different types of formal parameters*
- Function selection is based on the *number* and the *types of the actual parameters* at the places of invocation
- Function selection (*Overload Resolution*) is performed by the compiler
- Two functions having the same signature but different return types will result in a compilation error due to *attempt to re-declare*
- Overloading allows **Static Polymorphism**

Overload Resolution

Overload Resolution

Overload Resolution

- To resolve overloaded functions with one parameter
  - Identify the set of *Candidate Functions*
  - From the set of candidate functions identify the set of *Viable Functions*
  - Select the *Best viable function* through (*Order is important*)
    - ▷ *Exact Match*
    - ▷ *Promotion*
    - ▷ *Standard type conversion*
    - ▷ *User defined type conversion*

Overload Resolution: Exact Match

- *lvalue-to-rvalue conversion*: Read the value from an object
  - Most common
- *Array-to-pointer conversion*

  Definitions:
  ```
  int ar[10];
  void f(int *a);
  ```
  Call:
  ```
  f(ar)
  ```
- *Function-to-pointer conversion*

  Definitions:
  ```
  typedef int (*fp) (int);
  void f(int, fp);
  int g(int);
  ```
  Call:
  ```
  f(5, g)
  ```
- *Qualification conversion*
  - Converting pointer (only) to `const` pointer

Overload Resolution: Promotion & Conversion

- **Promotion**
  - Objects of an integral type can be converted to another wider integral type, that is, a type that can represent a larger set of values. This widening type of conversion is called *integral promotion*
  - C++ promotions are *value-preserving*, as the value after the promotion is guaranteed to be the same as the value before the promotion
  - Examples
    - ▷ `char` to `int`; `float` to `double`
    - ▷ `enum` to `int` / `short` / `unsigned int` / ...
    - ▷ `bool` to `int`

Overload Resolution: Promotion & Conversion

- **Standard Conversions**
  - *Integral conversions* between *integral types* – `char`, `short`, `int`, and `long` with or without qualifiers `signed` or `unsigned`
  - *Floating point Conversions* from *less precise floating type* to a *more precise floating type* like `float` to `double` or `double` to `long double`. Conversion can happen to a *less precise* type, if it is in a range representable by that type
  - *Conversions between integral and floating point types*: Certain expressions can cause objects of floating type to be converted to integral types, or vice versa. **May be dangerous!**
    - ▷ When an object of *integral type* is converted to a *floating type*, and the original value is not representable exactly, the result is either the next higher or the next lower representable value
    - ▷ When an object of *floating type* is converted to an *integral type*, the fractional part is truncated, or rounded toward zero. A number like 1.3 is converted to 1, and -1.3 is converted to -1
  - *Pointer Conversions*: Pointers can be converted during assignment, initialization, comparison, and other expressions
  - *Bool Conversion*: `int` to `bool` or vice versa based on the context

Example: Overload Resolution with one parameter

- In the context of a list of function prototypes:

```
int g(double);                  // F1
void f();                       // F2
void f(int);                    // F3
double h(void);                 // F4
int g(char, int);               // F5
void f(double, double = 3.4);   // F6
void h(int, double);            // F7
void f(char, char *);           // F8
```

  The call site to resolve is:

```
f(5.6);
```

- Resolution:
  - *Candidate functions* (**by name**): F2, F3, F6, F8
  - *Viable functions* (**by # of parameters**): F3, F6
  - *Best viable function* (**by type** `double` – **Exact Match**): F6

Example: Ambiguity in Overload Resolution

- Consider the overloaded function signatures:

```
int fun(float a) {...}              // Function 1
int fun(float a, int b) {...}       // Function 2
int fun(float x, int y = 5) {...} // Function 3

int main() {
    float p = 4.5, t = 10.5;
    int s = 30;

    fun(p, s); // CALL - 1
    fun(t);    // CALL - 2
    return 0;
}
```

- CALL - 1: Matches Function 2 & Function 3
- CALL - 2: Matches Function 1 & Function 3
- Results in ambiguity for both calls

Default Parameters in Overloading

Default Parameters in Overloading

Program 08.06/07: Default Parameter & Function Overload

- Compilers deal with *default parameters* as a special case of *function overloading*
- These need to be mixed carefully

Default Parameters

```
#include <iostream>
using namespace std;
int f(int a = 1, int b = 2);



int main() {
    int x = 5, y = 6;

    f();      // a = 1, b = 2
    f(x);     // a = x = 5, b = 2
    f(x, y); // a = x = 5, b = y = 6
}
```

- Function f has 2 parameters defaulted
- f can have 3 possible forms of call

Function Overload

```
#include <iostream>
using namespace std;
int f();
int f(int);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();      // int f();
    f(x);     // int f(int);
    f(x, y); // int f(int, int);
}
```

- Function f is overloaded with up to 2 parameters
- f can have 3 possible forms of call
- *No overload* here use *default parameters*. Can it?

Program 08.08: Default Parameter & Function Overload

- *Function overloading* can use *default parameter*
- However, *with default parameters*, the overloaded functions should *still be resolvable*

```
#include <iostream>
using namespace std;
// Overloaded Area functions
int Area(int a, int b = 10) { return (a * b); }
double Area(double c, double d) { return (c * d); }
int main() { int x = 10, y = 12, t; double z = 20.5, u = 5.0, f;
    t = Area(x);     // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // Area = 100

    t = Area(x, y);     // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // Area = 120

    f = Area(z, u); // Binds double Area(double, double)
    cout << "Area = " << f << endl;  // Area = 102.5

    f = Area(z); // Binds int Area(int, int = 10)
    cout << "Area = " << f << endl; // Area = 200

    // Un-resolvable between int Area(int a, int b = 10) and  double Area(double c, double d)
    f = Area(z, y); // Error: call of overloaded Area(double&, int&) is ambiguous
}
```

Program 08.09: Default Parameter & Function Overload

- Function overloading with default parameters may fail

```
#include <iostream>
using namespace std;
int f();
int f(int = 0);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();      // Error C2668: f: ambiguous call to overloaded function
              // More than one instance of overloaded function f
              // matches the argument list:
              //     function f()
              //     function f(int = 0)

    f(x);     // int f(int);
    f(x, y); // int f(int, int);

    return 0;
}
```

Module Summary

- Introduced the notion of Default parameters and discussed several examples
- Identified the necessity of function overloading
- Introduced static Polymorphism and discussed examples and restrictions
- Discussed an outline for Overload resolution
- Discussed the mix of default Parameters and function overloading

Programming in Modern C++

Module M09: Operator Overloading

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

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

Module Recap

- Introduced the notion of Default parameters and discussed several examples
- Identified the necessity of function overloading
- Introduced static Polymorphism and discussed examples and restrictions
- Discussed an outline for Overload resolution
- Discussed the mix of default Parameters and function overloading

Module Objectives

- Understand the Operator Overloading

Module Outline

1. Operators and functions
   - Difference
2. Operator Functions in C++
3. Operator Overloading
   - Advantages and Disadvantages
4. Examples of Operator Overloading
   - String: Concatenation
   - Enum: Changing the meaning of `operator+`
5. Operator Overloading Rules
6. Operator Overloading Restrictions
7. Module Summary

Operators and functions

Operators and functions

Operator & Function

- What is the difference between an *operator* & a *function*?

```
unsigned int Multiply(unsigned x, unsigned y) {
    int prod = 0;
    while (y-- > 0) prod += x;
    return prod;
}

int main() {
    unsigned int a = 2, b = 3;

    // Computed by ’*’ operator
    unsigned int c = a * b;          // c is 6

    // Computed by Multiply function
    unsigned int d = Multiply(a, b); // d is 6

    return 0;
}
```

- Same computation by an operator and a function

Difference between Operator & Functions

Operator

- Usually written in **infix** notation - at times in **prefix** or **postfix**
- Examples:

```
// Operator in-between operands
Infix: a + b; a ? b : c;
// Operator before operands
Prefix: ++a;
// Operator after operands
Postfix: a++;
```

- Operates on one or more operands, typically up to 3 (Unary, Binary or Ternary)
- Produces *one result*
- Order of operations is decided by *precedence* and *associativity*
- Operators are pre-defined

Function

- Always written in **prefix** notation
- Examples:

```
// Operator before operands
Prefix: max(a, b);
        qsort(int[], int, int,
            void (*)(void*, void*));
```

- Operates on zero or more arguments
- Produces *up to one result*
- Order of application is decided by *depth of nesting*
- Functions can be defined as needed

Operator Functions in C++

Operator Functions in C++

- C++ introduces a new keyword: `operator`
- Every operator is associated with an operator function that defines its behavior

Operator Expression	Operator Function
`a + b`	`operator+(a, b)`
`a = b`	`operator=(a, b)`
`c = a + b`	`operator=(c, operator+(a, b))`

- Operator functions are *implicit for predefined operators of built-in types* and *cannot be redefined*
- An operator function may have a signature as:

```
MyType a, b; // An enum or struct

MyType operator+(MyType, MyType); // Operator function

a + b // Calls operator+(a, b)
```

- C++ allows users to define an operator function and overload it

Operator Overloading

Operator Overloading

Operator Overloading

- **Operator Overloading** (also called *ad hoc polymorphism*), is a specific case of *polymorphism*, where different operators have different implementations depending on their arguments
- Operator overloading is generally defined by a programming language, For example, in C (and in C++), for `operator/`, we have:

Integer Division	Floating Point Division

`int i = 5, j = 2;`
`int k = i / j; // k = 2`
	`double i = 5, j = 2;`
	`double k = i / j; // k = 2.5`

- C does not allow programmers to overload its operators
- C++ allows programmers to overload its operators by using operator functions

Operator Overloading: Advantages and Disadvantages

- **Advantages**:
  - Operator overloading is *syntactic sugar*, and is used because it allows programming using notation nearer to the target domain
  - It also allows user-defined types a similar level of syntactic support as types built into a language
  - It is common in scientific computing, where it allows computing representations of mathematical objects to be manipulated with the same syntax as on paper
  - For example, if we build a `Complex` type in C and `a`, `b` and `c` are variables of `Complex` type, we need to code an expression

    ```
    a + b * c
    ```

    using functions to add and multiply Complex value as

    ```
    Add(a, Multiply(b, c))
    ```

    which is clumsy and non-intuitive
  - Using operator overloading we can write the expression with operators without having to use the functions

Operator Overloading: Advantages and Disadvantages

- **Disadvantages**
  - Operator overloading allows programmers to *reassign the semantics of operators* depending on the types of their operands. For example, for `int a, b`, an expression `a << b` shifts the bits in the variable `a` left by `b`, whereas `cout << a << b` outputs values of `a` and `b` to standard output (`cout`)
  - As operator overloading allows the programmer to change the usual semantics of an operator, it is a good practice to use operator overloading with care to maintain the *Semantic Congruity*
  - With operator overloading certain rules from mathematics can be *wrongly expected* or *unintentionally assumed*. For example, the commutativity of `operator+` (that is, `a + b == b + a`) is not preserved when we overload it to mean *string concatenation* as

    `"run" + "time" = "runtime"` ≠ `"timerun" = "time" + "run"`

    Of course, mathematics too has such deviations as multiplication is commutative for real and complex numbers but not commutative in matrix multiplication

Examples of Operator Overloading

Examples of Operator Overloading

Program 09.01: String Concatenation

Concatenation by string functions

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str; } String;
int main() { String fName, lName, name;
    fName.str = strdup("Partha ");
    lName.str = strdup("Das" );
    name.str = (char *) malloc( // Allocation
                   strlen(fName.str) +
                   strlen(lName.str) + 1);
    strcpy(name.str, fName.str);
    strcat(name.str, lName.str);
    cout << "First Name: " <<
            fName.str << endl;
    cout << "Last Name: " <<
            lName.str << endl;
    cout << "Full Name: " <<
            name.str << endl;
}
----------
First Name: Partha
Last Name: Das
Full Name: Partha Das
```

Concatenation operator

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str; } String;
String operator+(const String& s1, const String& s2) {
    String s;
    s.str = (char *) malloc(strlen(s1.str) +
             strlen(s2.str) + 1); // Allocation
    strcpy(s.str, s1.str); strcat(s.str, s2.str);
    return s;
}
int main() { String fName, lName, name;
    fName.str = strdup("Partha ");
    lName.str = strdup("Das");
    name = fName + lName; // Overloaded operator +
    cout << "First Name: " << fName.str << endl;
    cout << "Last Name: " << lName.str << endl;
    cout << "Full Name: " << name.str << endl;
}
----------
First Name: Partha
Last Name: Das
Full Name: Partha Das
```

Program 09.02: A new semantics for operator+

w/o Overloading +

```
#include <iostream>
using namespace std;
enum E { C0 = 0, C1 = 1, C2 = 2 };

int main() { E a = C1, b = C2;
    int x = -1;

    x = a + b; // operator + for int
    cout << x << endl;
}
----------
3
```

- Implicitly converts `enum E` values to `int`
- Adds by `operator+` of `int`
- Result is outside `enum E` range

Overloading `operator +`

```
#include <iostream>
using namespace std;
enum E { C0 = 0, C1 = 1, C2 = 2 };

E operator+(const E& a, const E& b) { // Overloaded operator +
    unsigned int uia = a, uib = b;
    unsigned int t = (uia + uib) % 3; // Redefined addition
    return (E) t;
}
int main() { E a = C1, b = C2;
    int x = -1;

    x = a + b; // Overloaded operator + for enum E
    cout << x << endl;
}
----------
0
```

- `operator +` is overloaded for `enum E`
- Result is a valid `enum E` value

Operator Overloading Rules

Operator Overloading Rules

Operator Overloading: Rules

- *No new operator* such as `operators**` or `operators<>` can be defined for overloading
- **Intrinsic properties** of the overloaded operator *cannot be changed*
  - Preserves *arity*
  - Preserves *precedence*
  - Preserves *associativity*
- These operators can be overloaded:

```
[] + - * / % ^ & | ~ ! = += -= *= /= %= ^= &= |=
<< >> >>= <<= == != < > <= >= && || ++ -- , ->* -> ( ) [ ]
```

- For *unary prefix operators*, use: `MyType& operator++(MyType& s1)`
- For *unary postfix operators*, use: `MyType operator++(MyType& s1, int)`
- The `operators::` (*scope resolution*), `operator.` (*member access*), `operator.*` (*member access through pointer to member*), `operator sizeof`, and `operator?:` (*ternary conditional*) *cannot be overloaded*
- The overloads of `operators&&`, `operator||`, and `operator,` (*comma*) *lose their special properties*: *short-circuit evaluation* and *sequencing*
- The overload of `operators->` must either return a raw pointer or return an object (by reference or by value), for which `operators->` is in turn overloaded

Operator Overloading Restrictions

Operator Overloading Restrictions

Overloading: Restrictions

`operator`	**Reason**
dot (`.`)	It will raise question whether it is for *object reference* or *overloading*
Scope Resolution (`::`)	It performs a (compile time) *scope resolution* rather than an *expression evaluation*
Ternary (`?:`)	Overloading `expr1? expr2: expr3` would not guarantee that *only one of* `expr2` and `expr3` was executed
`sizeof`	Operator `sizeof` cannot be overloaded because *built-in operations*, such as incrementing a pointer into an array *implicitly depends on it*
`&&` and `||`	In evaluation, the *second operand is not evaluated* if the result can be deduced *solely by evaluating the first operand*. However, this evaluation is not possible for overloaded versions of these operators
Comma (`,`)	This operator guarantees that the *first operand* is evaluated *before* the *second operand*. However, if the comma operator is overloaded, its operand evaluation depends on C++'s function parameter mechanism, which does not guarantee the order of evaluation
Ampersand (`&`)	The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares `operator&()` as a member function, then the behavior is undefined

Module Summary

- Introduced operator overloading with its advantages and disadvantages
- Explained the rules of operator overloading

Programming in Modern C++

Module M10: Dynamic Memory Management

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

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

Module Recap

- Introduced operator overloading with its advantages and disadvantages
- Explained the rules of operator overloading

Module Objectives

- Understand the dynamic memory management in C++

Module Outline

1. Dynamic Memory Management in C
    - malloc & free
2. Dynamic Memory Management in C++
    - new and delete operator
    - Dynamic memory allocation for Array
    - Placement new
    - Restrictions
3. Operator Overloading for Allocation and De-allocation
4. Module Summary

Dynamic Memory Management in C

Program 10.01/02: malloc() & free(): C & C++

C Program

```
#include <stdio.h>
#include <stdlib.h>


int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    printf("%d", *p);  // Prints: 5

    free(p);
}
```

C++ Program

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

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    cout << *p;  // Prints: 5

    free(p);
}
```

- Dynamic memory management functions in `stdlib.h` header for C (`cstdlib` header for C++)
- `malloc()` allocates the memory on heap or free store
- `sizeof(int)` needs to be provided
- Pointer to allocated memory returned as `void*` – needs cast to `int*`
- Allocated memory is released by `free()` from heap or free store
- `calloc()` and `realloc()` also available in both languages

Dynamic Memory Management in C++

Program 10.02/03: operator new & delete: Dynamic memory management in C++

- C++ introduces operators new and delete to dynamically allocate and de-allocate memory:

Functions malloc() & free()

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

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;
    cout << *p;  // Prints: 5

    free(p);
}
```

operator new & operator delete

```
#include <iostream>

using namespace std;

int main() {
    int *p = new int(5);

    cout << *p;  // Prints: 5

    delete p;
}
```

Functions malloc() & free()	operator new & operator delete
Function malloc() for allocation on heap	operator new for allocation on heap
sizeof(int) needs to be provided	*No size* specification needed, *type suffices*
Allocated memory returned as void*	Allocated memory returned as int*
Casting to int* *needed*	*No casting needed*
Cannot be initialized	*Can be initialized*
Function free() for de-allocation from heap	operator delete for de-allocation from heap
Library feature – header cstdlib needed	Core language feature – no header needed

Program 10.02/04: Functions: operator new() & operator delete()

- C++ also allows operator new() and operator delete() functions to dynamically allocate and de-allocate memory:

Functions malloc() & free()

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

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    cout << *p;  // Prints: 5

    free(p);
}
```

- Function malloc() for allocation on heap
- Function free() for de-allocation from heap

Functions operator new() & operator delete()

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

int main() {
    int *p = (int *)operator new(sizeof(int));
    *p = 5;

    cout << *p;  // Prints: 5

    operator delete(p);
}
```

- Function operator new() for allocation on heap
- Function operator delete() for de-allocation from heap

There is a major difference between operator new and function operator new(). We explore this angle later

Program 10.05/06: new[] & delete[]: Dynamically managed Arrays in C++

Functions malloc() & free()

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

int main() {
    int *a = (int *)malloc(sizeof(int)* 3);
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
             << a[i] << "     ";

    free(a);
}
-----
a[0] = 10     a[1] = 20     a[2] = 30
```

- Allocation by malloc() on heap

- # of elements implicit in size passed to malloc()
- Release by free() from heap

operator new[] & operator delete[]

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


int main() {
    int *a = new int[3];
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
             << a[i] << "     ";

    delete [] a;
}
-----
a[0] = 10     a[1] = 20     a[2] = 30
```

- Allocation by operator new[] (**different from** operator new) on heap
- # of elements explicitly passed to operator new[]
- Release by operator delete[] (**different from** operator delete) from heap

Program 10.07: Operator new(): Placement new in C++

```
#include <iostream>
using namespace std;
int main() { unsigned char buf[sizeof(int)* 2]; // Byte buffer on stack
    // placement new in buffer buf
    int *pInt = new (buf) int (3);
    int *qInt = new (buf+sizeof(int)) int (5);

    int *pBuf = (int *)(buf + 0);           // *pInt in buf[0] to buf[sizeof(int)-1]
    int *qBuf = (int *)(buf + sizeof(int)); // *qInt in buf[sizeof(int)] to buf[2*sizeof(int)-1]
    cout << "Buf Addr  Int Addr" << pBuf << "  " << pInt << endl << qBuf << "  " << qInt << endl;
    cout << "1st Int  2nd Int" << endl << *pBuf << "        " << *qBuf << endl;

    int *rInt = new int(7); // heap allocation
    cout << "Heap Addr  3rd Int" << endl << rInt << "   " << *rInt << endl;
    delete rInt;            // delete integer from heap
    // No delete for placement new
}
-----
Buf Addr  Int Addr
001BFC50  001BFC50
001BFC54  001BFC54
1st Int  2nd Int
3        5
Heap Addr  3rd Int
003799B8   7
```

- Placement operator new takes a buffer address to place objects
- These are not dynamically allocated on heap – may be allocated on stack or heap or static, wherever the buffer is located
- Allocations by Placement operator new must not be deleted

Mixing Allocators and De-allocators of C and C++

- Allocation and De-allocation must correctly match.
  - Do not free the space created by new using free()
  - And do not use delete if memory is allocated through malloc()

  These may results in memory corruption

Allocator	De-allocator
malloc()	free()
operator new	operator delete
operator new[]	operator delete[]
operator new()	No delete

- Passing NULL pointer to delete operator is secure
- Prefer to use only new and delete in a C++ program
- The new operator allocates exact amount of memory from Heap or Free Store
- new returns the given pointer type – no need to typecast
- new, new[ ] and delete, delete[] have separate semantics

Operator Overloading for Allocation and De-allocation

Operator Overloading for Allocation and De-allocation

Program 10.08: Overloading operator new and operator delete

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

void* operator new(size_t n) { // Definition of Operator new
    cout << "Overloaded new" << endl;
    void *ptr = malloc(n);     // Memory allocated to ptr. Can be done by function operator new()
    return ptr;
}
void operator delete(void *p) { // Definition of operator delete
    cout << "Overloaded delete" << endl;
    free(p);                     // Allocated memory released. Can be done by function operator delete()
}
int main() { int *p = new int; // Calling overloaded operator new
    *p = 30;          // Assign value to the location
    cout << "The value is :" << *p << endl;
    delete p;         // Calling overloaded operator delete
}
-----
Overloaded new
The value is :  30
Overloaded delete
```

- operator new overloaded
- The first parameter of overloaded operator new must be size_t
- The return type of overloaded operator new must be void*
- The first parameter of overloaded operator delete must be void*
- The return type of overloaded operator delete must be void
- More parameters may be used for overloading
- operator delete should not be overloaded (usually) with extra parameters

Program 10.09: Overloading operator new[] and operator delete[]

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

void* operator new [] (size_t os, char setv) { // Fill the allocated array with setv
    void *t = operator new(os);
    memset(t, setv, os);
    return t;
}
void operator delete[] (void *ss) {
    operator delete(ss);
}
int main() {
    char *t = new('#')char[10]; // Allocate array of 10 elements and fill with '#'

    cout << "p = " << (unsigned int) (t) << endl;
    for (int k = 0; k < 10; ++k)
        cout << t[k];

    delete [] t;
}
-----
p = 19421992
##########
```

- operator new[] overloaded with initialization
- The first parameter of overloaded operator new[] must be size_t
- The return type of overloaded operator new[] must be void*
- Multiple parameters may be used for overloading
- operator delete [] should not be overloaded (usually) with extra parameters

Module Summary

- Introduced `new` and `delete` for dynamic memory management in C++
- Understood the difference between `new`, `new[]` and `delete`, `delete[]`
- Compared memory management in C with C++
- Explored the overloading of `new`, `new[]` and `delete`, `delete[]` operators

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

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

Tutorial Objective

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

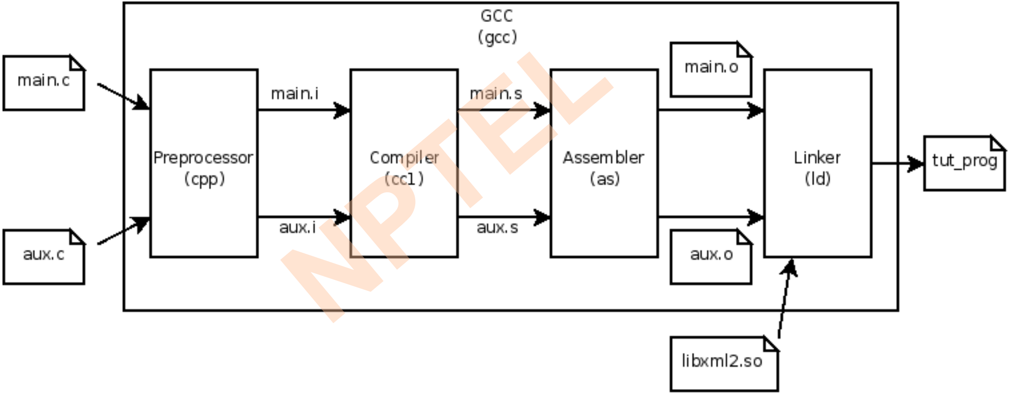
Tutorial Outline

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

Build Pipeline

Build Pipeline

Build Pipeline



Source: GNU Compiler Collection, Wikiwand Accessed 13-Sep-21

Build Pipeline

- 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

- **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 YoutTube
  - **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, or C++17
    - ▷ 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 (Accessed 13-Sep-21)
  - Code::Blocks is a free, open source cross-platform IDE that supports multiple compilers including GCC, Clang and Visual C++
  - Programiz Online Compiler supports C18 and C++14
  - OneCompiler supports C18 and C++17
- *For a compiler, you must know the language version you are compiling for - check to confirm*

What is GCC?

- **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++?

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: `g++ fileName.cpp -o binary`	Command to compile C program by gcc is: `gcc fileName.c -o binary -lstdc++`
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 `-lstdc++` 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

```
#define __GXX_WEAK__ 1
#define __cplusplus 1
#define __DEPRECATED 1
#define __GNUG__ 4
#define __EXCEPTIONS 1
#define __private_extern__ extern
```

Build with GCC: Options

[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

[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

[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

[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

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

- **Source file (user provided) extensions**

Extension	File Type	Extension	File Type
.c	C source code that must be preprocessed	**.cpp**, `.cc, .cp, .cxx` `.CPP, .c++, .C`	C++ source code that must be preprocessed

.h	C / C++ header file	`.H, .hp, .hxx, .hpp`
		`.HPP, .h++, .tcc`
			C++ header file

`.s`	Assembler code	`.S, .sx`	Assembler code that must be preprocessed

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

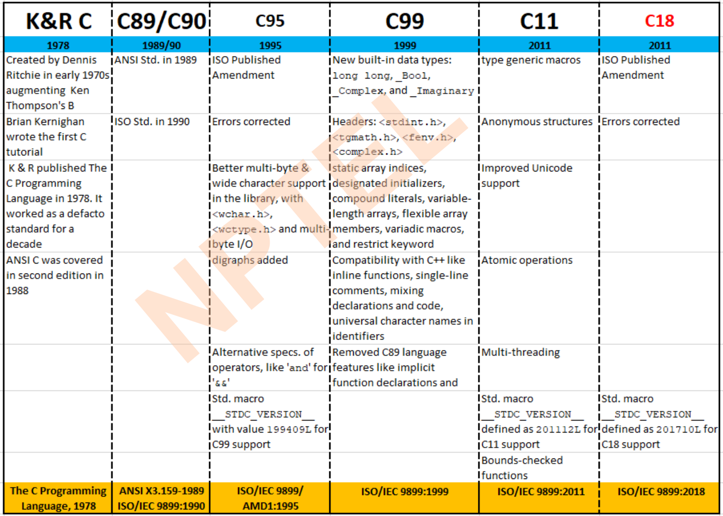
* We are going to follow the extensions marked in red

Source: 3.1 Option Summary and 3.2 Options Controlling the Kind of Output Accessed 13-Sep-21

C / C++ Dialects

C / C++ Dialects

C Dialects



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

- **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**

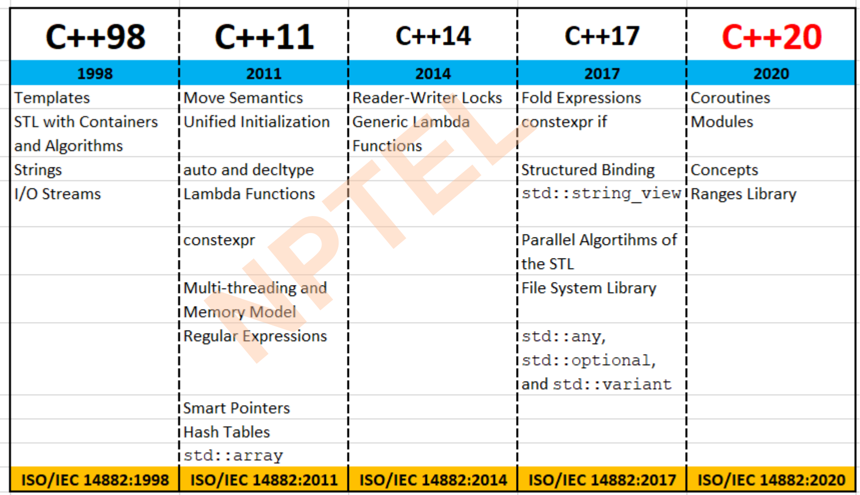
```
$ gcc -std=c99 "Check C Version.c"
C99

$ gcc "Check C Version.c"
C11

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

Default for this `gcc` is `C11`

C++ Standards



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

Fixes on C++98: C++03: ISO/IEC 14882:2003, 2003

Latest Version as of Sep-21: C++20: ISO/IEC 14882:2020, 2020

C++ Dialects: Checking for a dialect

- **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

Standard Library

What is Standard Library?

- 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

Component	Data Types, Manifest Constants, Macros, Functions, ...
stdio.h	Formatted and un-formatted file input and output including functions • printf, scanf, fprintf, fscanf, sprintf, sscanf, feof, etc.
stdlib.h	Memory allocation, process control, conversions, pseudo-random numbers, searching, sorting • malloc, free, exit, abort, atoi, strtold, rand, bsearch, qsort, etc.
string.h	Manipulation of C strings and arrays • strcat, strcpy, strcmp, strlen, strtok, memcpy, memmove, etc.
math.h	Common mathematical operations and transformations • cos, sin, tan, acos, asin, atan, exp, log, pow, sqrt, etc.
errno.h	Macros for reporting and retrieving error conditions through error codes stored in a static memory location called errno • EDOM (parameter outside a function's domain – sqrt(-1)), • ERANGE (result outside a function's range), or • EILSEQ (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

```
/* 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

C++ Standard Library: Common Library Components

Component	Data Types, Manifest Constants, Macros, Functions, Classes, ...
iostream	Stream input and output for standard I/O • cout, cin, endl, ..., etc.
string	Manipulation of string objects • Relational operators, IO operators, Iterators, etc.
memory	High-level memory management • Pointers: unique_ptr, shared_ptr, weak_ptr, auto_ptr, & allocator etc.
exception	Generic Error Handling • exception, bad_exception, unexpected_handler, terminate_handler, etc.
stdexcept	Standard Error Handling • logic_error, invalid_argument, domain_error, length_error, out_of_range, runtime_error, range_error, overflow_error, underflow_error, etc.
Adopted from C Standard Library	
cmath	Common mathematical operations and transformations • cos, sin, tan, acos, asin, atan, exp, log, pow, sqrt, etc.
cstdlib	Memory alloc., process control, conversions, pseudo-rand nos., searching, sorting • malloc, free, exit, abort, atoi, strtold, rand, bsearch, qsort, etc.

namespace std for C++ Standard Library

C Standard Library

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

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/o using

```
#include <iostream>


int main() {

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

    return 0;
}
```

W/ using

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

int main() {

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

    return 0;
}
```

Standard Library: C/C++ Header Conventions

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

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

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

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

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

This, however, is not preferred

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

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