C and C++

Intructors: Abir Das and Jibesh Patra

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

{abir, jibesh}@cse.iitkgp.ac.in

Slides heavily lifted from Programming in Modern C++ NPTEL Course

by Prof. Partha Pratim Das

Module Objectives

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

Note that here we are trying to understand the difference between the C-style of programming with the C++-style of programming, and how the C++ features make programming easier and less error-prone compared to its C equivalent. This is different from the compatibility issues between the two languages.

Program: Hello World

C Program

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

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

    return 0;
}
```

Hello World in C

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

C++ Program

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

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

    return 0;
}
```

Hello World in C++

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

Program: Add Two Numbers and Handling IO

C Program

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

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

    sum = a + b;

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

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

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

C++ Program

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

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

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

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

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

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

Program: Square Root of a number

C Program

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


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

    sqrt_x = sqrt(x);

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

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

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

C++ Program

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

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

    double sqrt_x = sqrt(x);

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

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

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

Program: Using bool

C Program

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

int main() {
    int x = TRUE;

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

bool is 1

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

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


int main() {
    bool x = true;

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

bool is 1

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

C++ Program

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

using namespace std;

int main() {
    bool x = true;

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

bool is 1

- No additional headers required

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

Program: Fixed Size Array

C Program

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

int main() {
    short age[4];

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

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

    return 0;
}
```

```
23 34 65 74
```

C++ Program

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

int main() {
    short age[4];

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

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

    return 0;
}
```

```
23 34 65 74
```

- No difference between arrays in C and C++

Arbitrary Size Array

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

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

Program: Fixed large array / vector

C (array & constant)

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

#define MAX 100

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

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

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

C++ (vector & constant)

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

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

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

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

Program: Dynamically managed array size

C Program

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


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

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

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

- malloc allocates space using sizeof

C++ Program

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

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

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

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

- resize fixes vector size at run-time

Strings in C and C++

String manipulations in C and C++:

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

Program: Concatenation of Strings

C Program

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


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

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

```
HELLO WORLD
```

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

C++ Program

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

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


    string str = str1 + str2;

    cout << str;
}
```

```
HELLO WORLD
```

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

More Operations on Strings

Further,

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

Program: Bubble Sort

C Program

```
#include <stdio.h>

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

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

```
    12 26 32 45 71
```

C++ Program

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

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

```
    12 26 32 45 71
```

- Implementation is same in both C and C++ apart from differences in header files

Program: Using sort from standard library

C Program (Desc order)

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

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

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

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

```
        71 45 32 26 12
```

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

C++ Program (Desc order)

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

    sort(data, data+5, compare);

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

```
    71 45 32 26 12
```

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

Stack in C

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

Program: Reversing a string in C

```
#include <stdio.h>

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

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

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

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

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

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

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

    int i, len = sizeof(str);

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

    printf("Reversed String: ");

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

```
Reversed String:  EDCBA
```

Understanding Stack in C++

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

Program: Reverse a String in C++

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


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

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

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

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

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

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

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

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

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

Data Structures / Containers in C++

- Like `Stack`, several other data structures are available in C++ standard library
- They are *ready-made* and *work like a data type*
- *Varied types of elements* can be used for C++ data structures
- **Data Structures** in C++ are commonly called **Containers**:
    - A container is a *holder object* that stores a *collection of other objects* (its elements)
    - The container
        - *manages the storage space* for its elements
        - provides member *functions to access* them
        - supports *iterators* - reference objects with similar properties to pointers
    - Many containers have several *member functions in common*, and *share functionalities* - easy to learn and remember

Module 06: Programming in Modern C++

Constants and Inline Functions

Instructors: Abir Das and jibesh Patra

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

{*abir, jibesh*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in C++

by **Prof. Partha Pratim Das**

Module Objectives

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

Module Outline

- `const`-ness and cv-qualifier
    - Notion of `const`
    - Advantages of `const`
        - Natural Constants – π, e
        - Program Constants – array size
        - Prefer `const` to `#define`
    - `const` and pointer
        - `const`-ness of pointer / pointee. How to decide?
    - Notion of `volatile`
- `inline` functions
    - Macros with params
        - Advantages
        - Disadvantages
    - Notion of `inline` functions
        - Advantages

Program 06.01: Manifest constants in C

- Manifest constants are defined by `#define`
- Manifest constants are replaced by CPP (C Pre-Processor)

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` **Manifest Constant**	**Using `const`** **Constant Variable**
- Is not type safe	- Has its type
- Replaced textually by CPP	- Visible to the compiler
- Cannot be *watched* in debugger	- Can be *watched* in debugger
- Evaluated as *many times as replaced*	- 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: Example

What will be the output of the following program:

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

int main() {
 const int a = 5;
 int *b;
 b = (int *) &a;
 *b = 10;
 cout << a << " " <<b<<" "<< &a <<" "<< *b <<"\n";
}
```

const and Pointers: Example

What will be the output of the following program:

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

int main() {
 const int a = 5;
 int *b;
 b = (int *) &a;
 *b = 10;
 cout << a << " " <<b<<" "<< &a <<" "<< *b <<"\n";
}
```

Standard g++ compiler prints: `5 0x16b58f4ec 0x16b58f4ec 10`
`b` actually points to `a`
But when accessed through `a` the compiler substitutes the constant expression Technically the behavior is **undefined**

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
- `const`
  - A `const` variable's value is set *only at initialization* – *can't be changed* afterwards
- `volatile`
  - In contrast, the value of a `volatile` variable can be modified by actions other than those in the user application.
  - Therefore, the volatile keyword is useful for declaring variables in shared memory that can be accessed by multiple processes for communication with interrupt service routines. It can be *changed by hardware, the kernel, another thread* etc.
  - When a name is declared as volatile, the compiler reloads the value from memory each time it is accessed by the program. This dramatically reduces the possible optimizations.
- `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).

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*
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

- Revisit manifest constants from C
- Understand `const`-ness, its use and advantages over manifest constants
- Understand the interplay of `const` and pointer
- Understand the notion and use of `volatile` data
- Revisit macros with parameters from C
- Understand `inline` functions and their advantages over macros
- Limitations of `inlineing`

Module 07: Programming in C++

Reference & Pointer

Intructors: Abir Das and Jibesh Patra

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

{*abir, jibesh*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

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

Module Outline

- Reference variable or Alias
  - Basic Notion
  - Call-by-reference in C++
- Example: Swapping two number in C
  - Using Call-by-value
  - Using Call-by-address
- Call-by-reference in C++ in contrast to Call-by-value in C
- Use of `const` in Alias / Reference
- Return-by-reference in C++ in contrast to Return-by-value in C
- Differences between References and Pointers

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

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
}
```

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

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 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

- **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

Pointers	**References**
• Refers to an *address (exposed)*	• Refers to an *address (hidden)*
• Pointers can point to `NULL`	• 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*	• *Does not allow* users to *operate on the address*
• diff pointers, increment, etc.	• 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