

# C and C++

Intructors: Abir Das and Sourangshu Bhattacharya

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

{*abir, sourangshu*}*@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**

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

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

    return 0;
}
```

**Hello World in C**

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

**C++ Program**

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

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

    return 0;
}
```

**Hello World in C++**

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

# 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**

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

int main() {
    int x = TRUE;

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

bool is 1

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

**C Program**

```c
// 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**

```cpp
// 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**

```c
// 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**

```cpp
// 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**

```c
// 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**

```cpp
// 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**

```c
// 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**

```cpp
// 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)**

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

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

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

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

```
71 45 32 26 12
```

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

**C++ Program (Desc order)**

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

    sort(data, data+5, compare);

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

```
71 45 32 26 12
```

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

# 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



```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;
}
```

```c
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++



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


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

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

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

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

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

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

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

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

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

# 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