Module 01: Programming in C++

Recap of C

Partha Pratim Das

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

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

Module Objectives

- Revisit the concepts of C language
- Revisit C Standard Library components
- Revisit the Organization and Build Process for C programs
- Create the foundation for the concepts of C++ with backward compatibility to C

Module Outline

- Recap of C features
  - Data types
  - Variables
  - Literals
  - Operators
  - Expressions
  - Statements
  - Control Constructs – Conditional Flow & Loops
  - Arrays
  - Structures & Unions
  - Pointers
  - Functions
  - Input / Output
- C Standard Library
- Source Organization for a C program
- Build Process

Module 01: Lecture 01

- Recap of C features
  - Data types
  - Variables
  - Literals
  - Operators
  - Expressions
  - Statements
  - Control Constructs – Conditional Flow & Loops

First C program

- Print "Hello World"

Source Program

```
#include <stdio.h>

int main() {

    printf("Hello World");
    printf("\n");

    return 0;
}
```

- `stdio.h` header included for input / output
- `main` function is used to start execution
- `printf` function is used to print the string "Hello World"

Data Types

Data types in C are used for declaring variables and deciding on storage and computations:

- **Built-in** / **Basic** data types are used to define raw data
  - `char`
  - `int`
  - `float`
  - `double`

  Additionally, C99 defines:
  - `bool`

  All data items of a given type has the same size (in bytes). The size is implementation-defined.
- **Enumerated Type** data are internally of `int` type and operates on a select subset.

Data Types

Data types in C further include:

- **void**: The type specifier `void` indicates no type.
- **Derived** data types include:
  - Array
  - Structure – `struct` & `union`
  - Pointer
  - Function
  - String – C-Strings are really not a type; but can be made to behave as such using functions from `<string.h>` in standard library
- **Type modifiers** include:
  - `short`
  - `long`
  - `signed`
  - `unsigned`

Variables

- A variable is a name given to a storage area
- Declaration of Variables:
  - Each variable in C has a specific type, which determines the size and layout of the storage (memory) for the variable
  - The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore

```
int    i, j, noOfData;
char   c, endOfSession;
float  f, velocity;
double d, dist_in_light_years;
```

Variables

- Initialization of Variables:
  - Initialization is setting an initial value to a variable at its definition

```
int    i = 10, j = 20, numberOfWorkDays = 22;
char   c = 'x';
float  weight = 4.5;
double density = 0.0;
```

Literals

- Literals refer to fixed values of a built-in type
- Literals can be of any of the basic data types

```
212     // (int) Decimal literal
0173    // (int) Octal literal
0b1010  // (int) Binary literal
0xF2    // (int) Hexadecimal literal
3.14    // (double) Floating-point literal
'x'     // (char) Character literal
"Hello" // (char *) String literal
```

- In C99, literals are constant values having `const` types as:

```
212     // (const int) Decimal literal
0173    // (const int) Octal literal
0b1010  // (const int) Binary literal
0xF2    // (const int) Hexadecimal literal
3.14    // (const double) Floating-point literal
'x'     // (const char) Character literal
"Hello" // (const char *) String literal
```

Operators

- An operator denotes a specific operation. C has the following types of operators:
  - Arithmetic Operators: + − * / % ++ −−
  - Relational Operators: == != > < >= <=
  - Logical Operators: && || !
  - Bit-wise Operators: & | ~ << >>
  - Assignment Operators: = += −= *= /= ···
  - Miscellaneous Operators: . , `sizeof` & * ?:
- **Arity of Operators:** Number of operand(s) for an operator
  - +, −, *, & operators can be *unary* (1 operand) or *binary* (2 operands)
  - ==, !=, >, <, >=, <=, &&, ||, +=, −=, *=, =, /=, &, |, <<, >> can work only as *b*inary (2 operands) operators
  - `sizeof` ! ~ ++ −− can work only as *unary* (1 operand) operators
  - ?: works as *t*ernary (3 operands) operator. The condition is the first operand and the `if` true logic and `if` false logic corresponds to the other two operands.

Operators

- **Operator Precedence:** Determines which operator will be performed first in a chain of different operators

  The precedence of all the operators mentioned above is in the following order: (left to right – Highest to lowest precedence)

  (), [], ++, –, + (unary), –(unary), !˜, *, &, sizeof, *, /, %, +, –, < <, > >, ==, !=, *=, =, /=, &, |, &&, | |, ?:, =, +=, –=, *=, =, /=, < <=, > >=

- **Operator Associativity:** Indicates in what order operators of equal precedence in an expression are applied

- Consider the expression a ˜ b ˜ c. If the operator ˜ has left associativity, this expression would be interpreted as (a ˜ b) ˜ c. If the operator has right associativity, the expression would be interpreted as a ˜ (b ˜ c).
  - Right-to-Left: ?:, =, +=, -=, *=, =, /=, <<=, >>=, –, +-, !˜, *, &, sizeof
  - Left-to-Right: *, /, %, +, -, <<, >>, ==, !=, *=, =, /=, &, |, &&, | |

Expressions

- Every expression has a value
  - A literal is an expression
  - A variable is an expression
  - One, two or three expression/s connected by an operator (of appropriate arity) is an expression
  - A function call is an expression
- Examples:
  - For

```
int i = 10, j = 20, k;
int f(int x, int y) { return x + y; }
```

  - Expression are:

```
2.5             // Value = 2.5
i               // Value 10
-i              // Value -10
i - j           // Value -10
k = 5           // Value 5
f(i, j)         // Value 30
i + j == i * 3 // Value true
(i == j)? 1: 2 // Value 2
```

Statement

- A statement is a command for a specific action. It has no value
  - A ; (semicolon) is a (null) statement
  - An expression terminated by a ; (semicolon) is a statement
  - A list of one or more statements enclosed within a pair of curly braces { and } or block is a compound statement
  - Control constructs like `if`, `if-else`, `switch`, `for`, `while`, `do-while`, `goto`, `continue`, `break`, `return` are statements
- Example: *Expression statements*

Expressions	Statements
`i + j`	`i + j;`
`k = i + j`	`k = i + j;`
`funct(i,j)`	`funct(i,j);`
`k = funct(i,j)`	`k = funct(i,j);`

- Example: *Compound statements*

```
{
    int i = 2, j = 3, t;

    t = i;
    i = j;
    j = t;
}
```

Control Constructs

- These statements control the flow based on conditions:
  - *Selection-statement*: `if`, `if-else`, `switch`
  - *Labeled-statement*: Statements labeled with identifier, `case`, or `default`
  - *Iteration-statement*: `for`, `while`, `do-while`
  - *Jump-statement*: `goto`, `continue`, `break`, `return`
- Examples:

```
if (a < b) {
    int t;

    t = a;
    a = b;
    b = t;
}
```

```
if (x < 5)
    x = x + 1;
else {
    x = x + 2;
    --y;
}
```

```
switch (i) {
    case 1: x = 5;
            break;
    case 3: x = 10;
    default: x = 15;
}
```

```
int sum = 0;
for(i = 0; i < 5; ++i) {
    int j = i * i;
    sum += j;
}
```

```
while (n) {
    sum += n;
    if (sum > 20)
        break;
    --n;
}
```

```
int f(int x, int y)
{
    return x + y;
}
```

Module 01: End of Lecture 01

- Recap of C features
  - Data types
  - Variables
  - Literals
  - Operators
  - Expressions
  - Statements
  - Control Constructs – Conditional Flow & Loops

Module 01: Lecture 02

- Recap of C features
  - Arrays
  - Structures
  - Unions
  - Pointers

Arrays

- An array is a collection of data items, all of the same type, accessed using a common name
  - Declare Arrays:

```
#define SIZE 10
int name[SIZE];     // SIZE must be an integer constant greater than zero
double balance[10];
```

  - Initialize Arrays:

```
int primes[5] = {2, 3, 5, 7, 11}; // Size = 5

int primes[]  = {2, 3, 5, 7, 11};
int sizeOfPrimes = sizeof(primes)/sizeof(int); // size is 5 by initialization

int primes[5] = {2, 3};           // Size = 5, last 3 elements set to 0
```

  - Access Array elements:

```
int primes[5] = {2, 3};
int EvenPrime = primes[0]; // Read 1st element
primes[2] = 5;             // Write 3rd element
```

  - Multidimensional Arrays:

```
int mat[3][4];

for(i = 0; i < 3; ++i)
    for(j = 0; j < 4; ++j)
        mat[i][j] = i + j;
```

Structures

- A structure is a collection of data items of different types. Data items are called *members*. The size of a structure is the sum of the size of its members.
  - Declare Structures:

```
struct Complex { // Complex Number
    double re;   // Real component
    double im;   // Imaginary component
} c;             // c is a variable of struct Complex type
printf("size = %d\n", sizeof(struct Complex)); // Prints: size = 16

typedef struct _Books {
    char  title[50];    // data member
    char  author[50];   // data member
    int   book_id;      // data member
} Books; // Books is an alias for struct _Books type
```

  - Initialize Structures:

```
struct Complex x = {2.0, 3.5}; // Both members
struct Complex y = {4.2};      // Only the first member
```

  - Access Structure members:

```
struct Complex x = {2.0, 3.5};
double norm = sqrt(x.re*x.re + x.im*x.im); // Using . (dot) operator

Books book;
book.book_id = 6495407;
strcpy(book.title, "C Programming");
```

Unions

- A union is a special structure that allocates memory only for the largest data member and holds only one member as a time
  - Declare Union:

```
typedef union _Packet { // Mixed Data Packet
    int    iData;       // integer data
    double dData;       // floating point data
    char   cData;       // character data
} Packet;
printf("size = %d\n", sizeof(Packet)); // Prints: size = 8
```

  - Initialize Union:

```
Packer p = {10}; // Initialize only with a value of the type of first member
printf("iData = %d\n", p.iData);   // Prints: iData = 10
```

  - Access Union members:

```
p.iData = 2;
printf("iData = %d\n", p.iData);   // Prints: iData = 2
p.dData = 2.2;
printf("dData = %lf\n", p.dData);  // Prints: dData = 2.200000
p.cData = 'a';
printf("cData = %c\n", p.cData);   // Prints: cData = a

p.iData = 97;
printf("iData = %d\n", p.iData);   // Prints: iData = 97
printf("dData = %lf\n", p.dData);  // Prints: dData = 2.199999
printf("cData = %c\n", p.cData);   // Prints: cData = a
```

Pointers

- A pointer is a variable whose value is a memory address
- The type of a pointer is determined by the type of its pointee

```
int    *ip;   // pointer to an integer
double *dp;   // pointer to a double
float  *fp;   // pointer to a float
char   *ch    // pointer to a character
```

- Using a pointer:

```
int main() {
    int  i = 20;   // variable declaration
    int  *ip;      // pointer declaration
    ip = &i;       // store address of i in pointer

    printf("Address of variable: %p\n", &i); // Prints: Address of variable : 00A8F73C

    printf("Value of pointer: %p\n", ip);    // Prints: Value of pointer : 00A8F73C

    printf("Value of pointee: %d\n", *ip);   // Prints: Value of pointee : 20

    return 0;
}
```

Pointers

• Pointer-Array Duality

```
int a[] = {1, 2, 3, 4, 5};
int *p;

p = a;
printf("a[0] = %d\n", *p);     // a[0] = 1
printf("a[1] = %d\n", *++p);   // a[1] = 2
printf("a[2] = %d\n", *(p+1)); // a[2] = 3

p = &a[2];
*p = -10;
printf("a[2] = %d\n", a[2]);   // a[2] = -10
```

• malloc-free

```
int *p = (int *)malloc(sizeof(int));

*p = 0x8F7E1A2B;
printf("%X\n", *p);   // 8F7E1A2B

unsigned char *q = p;
printf("%X\n", *q++); // 2B
printf("%X\n", *q++); // 1A
printf("%X\n", *q++); // 7E
printf("%X\n", *q++); // 8F

free(p);
```

• Pointer to a structure

```
struct Complex { // Complex Number
    double re;   // Real component
    double im;   // Imaginary component
} c = { 0.0, 0.0 };

struct Complex *p = &c;

(*p).re = 2.5;
p->im = 3.6;

printf("re = %lf\n", c.re); // re = 2.500000
printf("im = %lf\n", c.im); // im = 3.600000
```

• Dynamically allocated arrays

```
int *p = (int *)malloc(sizeof(int)*3);

p[0] = 1; p[1] = 2; p[2] = 3;

printf("p[1] = %d\n", *(p+1)); // p[1] = 2
free(p);
```

Module 01: End of Lecture 02

- Recap of C features
  - Arrays
  - Structures
  - Unions
  - Pointers

Module 01: Lecture 03

- Recap of C features
  - Functions
  - Input / Output
- C Standard Library
- Source Organization for a C program
- Build Process

Functions

- A function performs a specific task or computation
  - Has 0, 1, or more parameters / arguments. Every argument has a type (`void` for no argument)
  - May or may not return a result. Return value has a type (`void` for no result)
  - Function declaration:

```
// Function Prototype / Header / Signature
// Name of the function: funct
// Parameters: x and y. Types of parameters: int
// Return type: int

int funct(int x, int y);
```

  - Function definition:

```
// Function Implementation
int funct(int x, int y)

// Function Body
{
    return (x + y);
}
```

Functions

- **Call-by-value** mechanism for passing arguments. The value of an actual parameter copied to the formal parameter
- **Return-by-value** mechanism to return the value, if any.

```
int funct(int x, int y) {
    ++x; ++y;                // Formal parameters changed
    return (x + y);
}

int main() {
    int a = 5, b = 10, z;

    printf("a = %d, b = %d\n", a, b); // prints: a = 5, b = 10

    z = funct(a, b); // function call by value
                     // a copied to x. x becomes 5
                     // b copied to y. y becomes 10
                     // x in funct changes to 6  (++x)
                     // y in funct changes to 11 (++y)
                     // return value (x + y) copied to z

    printf("funct = %d\n", z); // prints: funct = 17

    // Actual parameters do not change on return (call-by-value)
    printf("a = %d, b = %d\n", a, b); // prints: a = 5, b = 10

    return 0;
}
```

Functions

- A function may be recursive (call itself)
  - Has recursive step/s
  - Has exit condition/s
- Example:

```
// Factorial of n
unsigned int factorial(unsigned int n) {
    if (n > 0)
        return n * factorial(n - 1); // Recursive step
    else
        return 1;                    // Exit condition
}

// Number of 1's in the binary representation of n
unsigned int nOnes(unsigned int n) {
    if (n == 0)
        return 0; // Exit condition
    else // Recursive steps
        if (n % 2 == 0)
            return nOnes(n / 2);
        else
            return nOnes(n / 2) + 1;
}
```

Function pointers


```
#include <stdio.h>
struct GeoObject {
    enum { CIR = 0, REC, TRG } gCode;
    union {
        struct Cir { double x, y, r; } c;
        struct Rec { double x, y, w, h; } r;
        struct Trg { double x, y, b, h; } t;
    };
};

typedef void(*DrawFunc) (struct GeoObject);

void drawCir(struct GeoObject go) {
    printf("Circle: (%lf, %lf, %lf)\n",
        go.c.x, go.c.y, go.c.r); }

void drawRec(struct GeoObject go) {
    printf("Rect: (%lf, %lf, %lf, %lf)\n",
        go.r.x, go.r.y, go.r.w, go.r.h); }

void drawTrg(struct GeoObject go) {
    printf("Triag: (%lf, %lf, %lf, %lf)\n",
        go.t.x, go.t.y, go.t.b, go.t.h); }
```

```
DrawFunc DrawArr[] = { // Array of func. ptrs
    drawCir, drawRec, drawTrg };

int main() {
    struct GeoObject go;

    go.gCode = CIR;
    go.c.x = 2.3; go.c.y = 3.6;
    go.c.r = 1.2;
    DrawArr[go.gCode](go); // Call by ptr

    go.gCode = REC;
    go.r.x = 4.5; go.r.y = 1.9;
    go.r.w = 4.2; go.r.h = 3.8;
    DrawArr[go.gCode](go); // Call by ptr

    go.gCode = TRG;
    go.t.x = 3.1; go.t.y = 2.8;
    go.t.b = 4.4; go.t.h = 2.7;
    DrawArr[go.gCode](go); // Call by ptr

    return 0;
}
```

```
Circle: (2.300000, 3.600000, 1.200000)
Rect: (4.500000, 1.900000, 4.200000, 3.800000)
Triag: (3.100000, 2.800000, 4.400000, 2.700000)
```

Input / Output

- `int printf(const char *format, ...)` writes to `stdout` by the format and returns the number of characters written
- `int scanf(const char *format, ...)` reads from `stdin` by the format and returns the number of characters read
- Use %s, %d, %c, %lf, to print/scan string, `int`, `char`, `double`

```
#include <stdio.h>

int main() {

    char str[100];
    int i;

    printf("Enter a value :");              // prints a constant string

    scanf("%s %d", str, &i);                // scans a string value and an integer value

    printf("You entered: %s %d ", str, i); // prints string and integer

    return 0;
}
```

Input / Output

- To write to or read from file:

```
#include <stdio.h>

int main() {

    FILE *fp = NULL;
    int i;

    fp = fopen("Input.dat", "r");
    fscanf(fp, "%d", &i);           // scan from Input.dat
    fclose(fp);

    fp = fopen("Output.dat", "w");
    fprintf("%d^2 = %d\n", i, i*i); // prints to Output.dat
    fclose(fp);

    return 0;
}
```

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.

Source Organization for a C program

Header Files

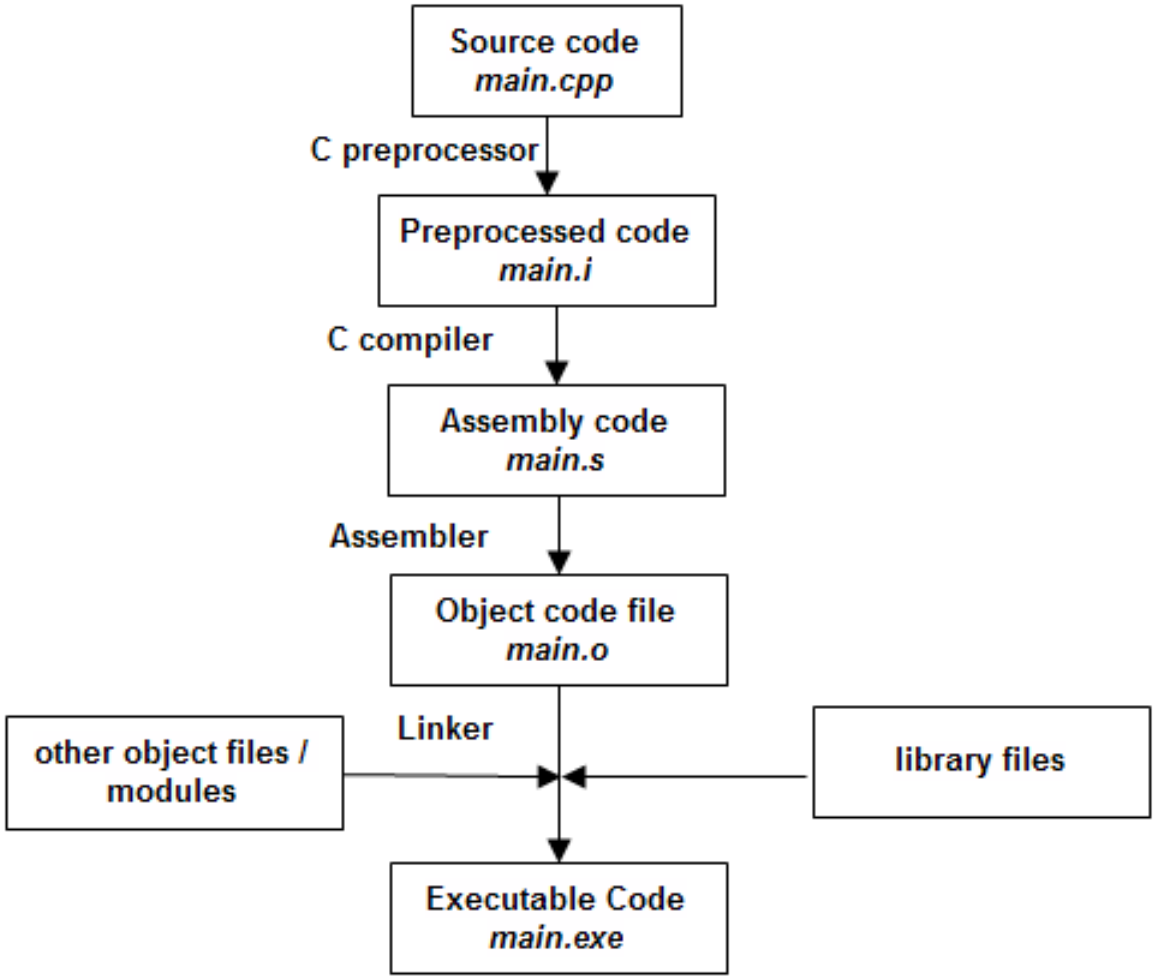
- A header file has extension `.h` and contains C function declarations and macro definitions to be shared between several source files
- There are two types of header files:
  - Files that the programmer writes
  - Files from standard library
- Header files are included using the `#include` pre-processing directive
  - `#include <file>` for system header files
  - `#include "file"` for header files of your own program

Source Organization for a C program

- Example:

```
// Solver.h --  Header files
int quadraticEquationSolver(double, double, double, double*, double*);

// Solver.c -- Implementation files
#include "Solver.h"

int quadraticEquationSolver(double a, double b, doublec , double* r1, double* r2) {
    // ...
    // ...
    // ...
    return 0;
}

// main.c  -- Application files
#include "Solver.h"

int main() {
    double a, b, c;
    double r1, r2;

    int status = quadraticEquationSolver(a, b, c, &r1, &r2);

    return 0;
}
```

Build Flow



Build Process

- C Pre-processor (CPP) substitutes and includes functions, headers and macros before compilation

```
int sum(int, int);
int main() {
    int a = sum(1,2);
    return a;
}
```

- The compiler translates the pre-processed C code into assembly language, which is a machine level code that contains instructions that manipulate the memory and processor directly
- The linker links our program with the pre-compiled libraries for using their functions
- In the running example, `function.c` and `main.c` are first compiled and then linked

```
int sum(int a,int b) { return a+b; }

int main() {
    int a = sum(1,2);  // as files are linked, uses functions directly
    return a;
}
```

Tools

- Development IDE: Code::Blocks 16.01
- Compiler: -std=c++98 and -std=c99

References

- Kernighan, Brian W., and Dennis M. Richie. The C Programming Language. Vol. 2. Englewood Cliffs: Prentice-Hall, 1988.
- King, Kim N., and Kim King. C programming: A Modern Approach. Norton, 1996.

Module Summary

- Revised the concept of variables and literals in C
- Revised the various data types and operators of C
- Re-iterated through the control constructs of C
- Re-iterated through the concepts of functions and pointers of C
- Re-iterated through the program organization of C and the build process.

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655

Module 02: Programming in C++

Programs with IO & Loop

Partha Pratim Das

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

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

Module Objectives

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

Module Outline

- Contrast differences between C and C++ programs for:
  - I/O
  - Variables
  - Using math library
  - Standard Library – Headers
  - Loop
  - `bool` type

Program 02.01: Hello World

C Program	C++ Program

`// FileName:HelloWorld.c:`
`#include <stdio.h>`
 	
`int main() {`
 	
`  printf("Hello World in C");`
`  printf("\n");`
 	
`  return 0;`
`}`
	`// FileName:HelloWorld.cpp:`
	`#include <iostream>`
	
	`int main() {`
	
	`  std::cout << "Hello World in C++";`
	`  std::cout << std::endl;`
	
	`  return 0;`
	`}`

Hello World in C	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	• 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 02.02: Add two numbers

C Program

```
// FileName: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);

    return 0;
}
```

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 (C89)
- Formatting (%d) needed for variables

C++ Program

```
// FileName: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;

    return 0;
}
```

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
- Formatting is derived from type (int) of variables

Program 02.03: Square Root of a number

C Program

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


int main() {
    double x;
    double 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);

    return 0;
}
```

```
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

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

int main() {
    double x;


    cout << "Input number:" << endl;
    cin >> x;

    double sqrt_x =     // Declaration of sqrt_x
        sqrt(x);

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

    return 0;
}
```

```
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)

namespace std for C++ Standard Library

C Standard Library	C++ Standard Library
• All names are global	• All names are within `std` namespace
• `stdout`, `stdin`, `printf`, `scanf`	• `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 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>`

- Any C standard library header is to be used in C++ with a prefix 'c' and without the `.h`. These symbols will be in `std` namespace. Like:

```
#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.**

Program 02.04: Sum n natural numbers

C Program

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


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

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

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

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

    return 0;
}
```

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

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

C++ Program

```
// FileName:Sum_n_c++.cpp:
#include <iostream>
using namespace std;

int main() {
    int n;

    int sum = 0;

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

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

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

    return 0;
}
```

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

- `i` declared locally in `for` loop

Program 02.05: Using bool

C Program

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

int main() {
    int x = TRUE;

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

    return 0;
}
```

bool is 1

- Using `int` and `#define` for bool
- May use `_Bool` (C99)

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


int main() {
    bool x = true;

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

    return 0;
}
```

bool is 1

- `stdbool.h` included for bool
- `_Bool` type & macros (C99):
  `bool` which expands to `_Bool`
  `true` which expands to `1`
  `false` which expands to `0`

C++ Program

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

using namespace std;

int main() {
    bool x = true;

    cout <<
        "bool is " << x;

    return 0;
}
```

bool is 1

- No additional headers required

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

Module Summary

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

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655

Module 03: Programming in C++

Arrays and Strings

Partha Pratim Das

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

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

Module Objectives

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

Module Outline

- Arrays and Vectors
  - Fixed size arrays – in C and C++
  - Arbitrary size arrays – in C and C++
  - `vectors` in C++
- Strings in C and C++
  - `string` functions in C and C++
  - `string` type in C++
  - String manipulation in C++

Program 03.01: Fixed Size Array

C Program

```
// File Name: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

```
//FileName: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 03.02: Fixed size large array in C

Hard-coded

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


int main() {
    int arr[100], sum = 0, i;

    printf("Enter no. of elements: ");
    int count;
    scanf("%d", &count);

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

    return 0;
}
```

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

- Hard-coded size

Using manifest constant

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

int main() {
    int arr[MAX], sum = 0, i;

    printf("Enter no. of elements: ");
    int count;
    scanf("%d", &count);

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

    return 0;
}
```

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

- Size by manifest constant

Program 03.03: Fixed large array / vector

C (array & constant)

```
// FileName:Array_Macro.c:
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

int main() {
    int arr[MAX], sum = 0, i;

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

```
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)

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

int main() {
    vector<int> arr(MAX); // Define-time size

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

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

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

Program 03.04: Dynamically managed array size

C Program

```
// FileName: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);
    return 0;
}
```

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

- malloc allocates space using sizeof

C++ Program

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

int main() {
     cout << "Enter the no. of elements: ";
     int count, j, 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;
     return 0;
}
```

```
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) behaves like a built-in type

Program 03.05: Concatenation of Strings

C Program

```
// FileName: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);

    return 0;
}
```

```
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

```
// FileName: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;

    return 0;
}
```

```
HELLO WORLD
```

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

More 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

Module Summary

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

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655

Module 04: Programming in C++

Sorting and Searching

Partha Pratim Das

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

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

Module Objectives

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

Module Outline

- Sorting in C and C++
  - Bubble Sort
  - Using Standard Library
- Searching in C and C++
  - Using Standard Library
- `algorithm` Library

Program 04.01: Bubble Sort

C Program

```
// FileName:Bubble_Sort.c:
#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]);

    return 0;
}
```

```
12 26 32 45 71
```

C++ Program

```
// FileName:Bubble_Sort.cpp:
#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] << " ";

     return 0;
}
```

```
12 26 32 45 71
```

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

Program 04.02: Using sort from standard library

C Program (Desc order)

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


// compare Function Pointer
int compare(const void *a, const void *b) {
    return (*(int*)a < *(int*)b);
}

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]);

    return 0;
}
```

```
 71 45 32 26 12
```

- sizeof int, array passed in qsort

C++ Program (Desc order)

```
// FileName:Algorithm_Cust_c++.cpp:
#include <iostream>
#include <algorithm>
using namespace std;

// compare Function Pointer
bool compare (int i, int j) {
    return (i > j);
}

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] << " ";

    return 0;
}
```

```
71 45 32 26 12
```

- Size need not be passed.

Program 04.03: Using default sort of algorithm

C++ Program

```
// FileName:Algorithm_Cust_c++.cpp:
#include <iostream>
#include <algorithm>
using namespace std;

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

    sort (data, data+5);

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

    return 0;
}
```

```
12 26 32 45 71
```

- Sort using the default sort function of algorithm library which does the sorting in ascending order only.

Program 04.04: Binary Search

C Program

```
// FileName:Binary_Search.c:
#include <stdio.h>
#include <stdlib.h>


// compare Function Pointer
int compare (const void * a, const void * b) {
    if ( *(int*)a <  *(int*)b ) return -1;
    if ( *(int*)a == *(int*)b ) return 0;
    if ( *(int*)a >  *(int*)b ) return 1;
}

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

    if (bsearch (&key, data, 5,
            sizeof(int), compare))
        cout << "found!\n";
    else
        cout << "not found.\n";

    return 0;
}
```

```
found!
```

C++ Program

```
// FileName:Binary_Search_c++.cpp:
#include <iostream>
#include <algorithm>
using namespace std;


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

    if (binary_search (data, data+5, key))

        cout << "found!\n";
    else
        cout << "not found.\n";

    return 0;
}
```

```
found!
```

The algorithm Library

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

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

Program 04.05: replace and rotate functions

Replace

```
// FileName:Replace.cpp:
#include <iostream>
#include <algorithm>
using namespace std;

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

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

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

    return 0;
}
```

```
1 2 2 4 5
```

- 3rd element replaced with 2

Rotate

```
// FileName:Rotate.cpp:
#include <iostream>
#include <algorithm>
using namespace std;

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

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

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

    return 0;
}
```

```
3 4 5 1 2
```

- Array circular shifted around 3rd element.

Module Summary

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

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655

Module 05: Programming in C++

Stack and its Applications

Partha Pratim Das

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

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

Module Objectives

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

Module Outline

- Stack in C
  - Reverse a String
  - Evaluate a Postfix Expression
- Stack in C++
  - Reverse a String
  - Evaluate a Postfix Expression

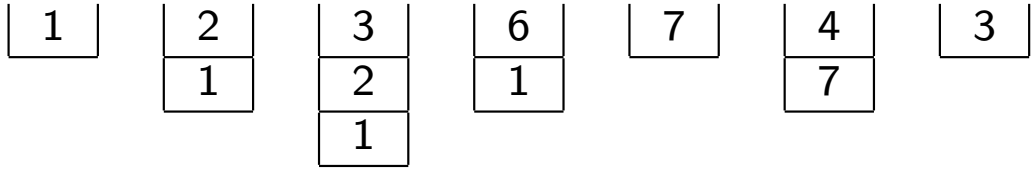
Understanding Stack in C

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

Common C programs using stack

Some common C programs that use stack:

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



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

Program 05.01: Reversing a string


```
// FileName: Reverse_String.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;
    }
}
```

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

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

    int i, len = sizeof(str);

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

    printf ("Reversed String: ");

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

Reversed String: EDCBA

Program 05.02: Postfix Expression Evaluation


```
// FileName: PostFix_Evaluation.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;
   }
}
```

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

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

    int i, op1, op2;

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

Evaluation 3

Understanding Stack in C++

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

Program 05.03: Reverse a String in C++


```
// FileName: Reverse_String_c++.cpp
#include<iostream>
#include<string.h>
#include<stack>
using namespace std;

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

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

    cout << "Reversed String: ";

    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }

    return 0;
}
```

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

```
// FileName: Reverse_String.c


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

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

    printf ("Reversed String: ");

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

    return 0;
}
```

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

Program 05.04: Postfix Evaluation in C++


```
// FileName:Postfix_Evaluation_c++.cpp
#include <iostream>
#include <stack>
using namespace std;

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

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

Module Summary

- C++ standard library provides ready-made stack. It works like a data type
- Any type of element can be used for C++ stack
- Similar containers as available in C++ standard library include:
  - `queue`
  - `deque`
  - `list`
  - `map`
  - `set`
  - ... and more

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655