C Language

Contents

[Basic 3](#_Toc476123192)

[Introduction 3](#_Toc476123193)

[C vs C++ 3](#_Toc476123194)

[Program runs in C but not in C++ 4](#_Toc476123195)

[Macros and Preprocessors 4](#_Toc476123196)

[Facts about Preprocessors 4](#_Toc476123197)

[Data type 6](#_Toc476123198)

[Enum, Struct and Union 7](#_Toc476123199)

[Enumeration (or enum) 7](#_Toc476123200)

[ENUM ADVANTAGES over Macro 7](#_Toc476123201)

[Union 8](#_Toc476123202)

[Structure Member Alignment, Padding and Data Packing 8](#_Toc476123203)

[Bit Fields 9](#_Toc476123204)

[interesting facts about bit fields 9](#_Toc476123205)

[Little endian or big endian 11](#_Toc476123206)

[Control Statements 11](#_Toc476123207)

[Switch Case 11](#_Toc476123208)

[LOOPS 11](#_Toc476123209)

[For 11](#_Toc476123210)

[While 11](#_Toc476123211)

[DO while 12](#_Toc476123212)

# Basic

## Introduction

* C is a procedural programming language.
* Initially developed by Dennis Ritchie between 1969 and 1973.
* C++ is nearly a superset of C language.
* C++ was developed by Bjarne Stroustrup at Bell Labs since 1979
* Main features of C
  1. Low-level access to memory
  2. Simple set of keywords, and
  3. Clean style
* Suitable Language for
  1. Operating system
  2. Compiler development
* Free compliers
  1. [Code Blocks](http://www.codeblocks.org/downloads/26) and [Dev-CPP](http://www.bloodshed.net/download.html)
  2. [ideone.com](http://ideone.com/) or [codepad.org](http://codepad.org/)

|  |
| --- |
| * Main function   1. argc 🡪 "argument count"   2. argv 🡪 "argument vector" which is a one-dimensional array of strings   3. These **arguments passed to a program through the** command line   4. argv[0] is location on exe file   **int** main(**int** argc, **char\*** argv[]) { /\* ... \*/ }  ***int*** *main() { /\* ... \*/ }* |

* Four phases for a C/C++ program to become an executable

1. **Pre-processing**
   1. Removal of Comments
   2. Expansion of Macros
   3. Expansion of the included files
2. **Compilation**
   1. The next step is to compile ***filename.i*** and produce an intermediate compiled output file ***filename.s****.* This file is in assembly level instructions.
3. **Assembly**
   1. In this phase the ***filename.s*** is taken as input and turned into ***filename.o*** by assembler. This file contains machine level instructions.
4. **Linking**
   1. This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments. This task can be easily verified by using ***$size filename.o*** and ***$size filename***. Through these commands, we know that how output file increases from an object file to an executable file.

# C vs C++

## Program runs in C but not in C++

* In C, if a function signature doesn’t specify any argument, it means that the function can be called with any number of parameters or without any parameters.

// Program 1 (**Compiles and runs fine in C, but not in C++**)

**void** fun() {  }

**int** main(void){

    fun(10, "GfG", "GQ");

**return** 0;

}

// Program 2 (**Compiles and runs fine in C, but not in C++**)

**int** main(){

**static** **int** i = 5;

    if (--i){

        printf("%d ", i);

        main(10); // C can accept any number of args

    }

}

// Program 3 (**Fails in compilation in both C and C++)**

**void** fun(**void**) {  }

**int** main(**void**){// specifies that main can only be called w/o any parameter.

    fun(10, "GfG", "GQ");

**return** 0;

}

// Program 4 (**Fails in compilation in both C and C++)**

**int** main(void){// compiler except only without parameter

**static int** i = 5;

**if** (--i){

        printf("%d ", i);

        main(10);

    }

}

# Macros and Preprocessors

1. Lines that start with # are *preprocessing directives.*
2. A line containing only # is also a preprocessing directive, but it has no effect.
3. **#include**, **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif**, **#define**, **#undef**, **#line**, **#error**, and **#pragma** are all preprocessing directives.
4. **#define MAX 8** is a **preprocessing directive**, it is not a macro. MAX is a macro.

## Facts about Preprocessors

1. ***#include***directive

The contents of **included header file (after preprocessing) are copied** to the current file.

**#include**<stdio.h>

1. ***#define***for a constant,

Defined constant is searched and matching tokens are **replaced with the given expression**.

**#define** MAX 100

1. The macros can take function like arguments; the arguments are not checked for data type.

**#define** INCREMENT(x) ++x

**int** main(){

**char** \*ptr = "GeeksQuiz";

**int** x = 10;

    printf("%s  ", INCREMENT(ptr));

    printf("%d", INCREMENT(x));

    return 0;

}

// Output: eeksQuiz 11

1. The macro arguments are not evaluated before macro expansion and they with arguments should be avoided as they cause problems sometimes.

**#define** MULTIPLY(a, b) a\*b

**int** main(){

    // The macro is expended as 2 + 3 \* 3 + 5, not as 5\*8

    printf("%d", MULTIPLY(2+3, 3+5));

    return 0;

}

// Output: 16

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

**int** main(){

  int x = 36/**SQUARE**(6); // Expended as 36/6\*6

  printf("%d", x);

  return 0;

}

// Output: 36

1. The tokens passed to macros can be concatenated using operator ## called **Token-Pasting** operator.

**#define** MERGE (a, b) a##b

**int** main(){

    printf("%d ", MERGE(12, 34));

}

// Output: 1234

1. A token passed to macro can be converted to a sting literal by using # before it.

**#define** GET(a) **#**a

**int** main(){

    // GeeksQuiz is changed to "GeeksQuiz"

    printf("%s", GET(GeeksQuiz));

}

// Output: GeeksQuiz

1. The macros can be written in multiple lines using ‘\’.

**#define** PRINT(i, limit) while (i < limit) **\**

                        { **\**

                            printf("GeeksQuiz "); **\**

                            i++; **\**

                        }

1. Preprocessors also support if-else directives which are typically used for conditional compilation.

**#if** VERBOSE >= 2

  printf("Trace Message");

**#endif**

1. A header file may be included more than one time directly or indirectly, this leads to problems of redeclaration of same variables/functions. To avoid this problem, directives like defined, ifdef and ifndef are used.

**#ifndef** \_MYHEADERFILE\_H\_

**#define** \_MYHEADERFILE\_H\_

// code for Header

**#endif**

1. There are some standard macros which can be used to print program file (\_\_FILE\_\_), Date of compilation (\_\_DATE\_\_), Time of compilation (\_\_TIME\_\_) and Line Number in C code (\_\_LINE\_\_)

**int** main(){

   printf("Current File :%s\n", \_\_FILE\_\_ );

   printf("Current Date :%s\n", \_\_DATE\_\_ );

   printf("Current Time :%s\n", \_\_TIME\_\_ );

   printf("Line Number :%d\n", \_\_LINE\_\_ );

   return 0;

}

/\* Output:

Current File :C:\Users\GfG\Downloads\deleteBST.c

Current Date :Feb 15 2014

Current Time :07:04:25

Line Number :8 \*/

# Data type

<http://quiz.geeksforgeeks.org/c-language-2/data-types/>

32 bit GCC compiler

Data Type Memory (bytes) Range Format Specifier

short int 2 -32,768 to 32,767 %hd

unsigned short int 2 0 to 65,535 %hu

unsigned int 4 0 to 4,294,967,295 %u

int 4 -2,147,483,648 to 2,147,483,647 %d

long int 4 -2,147,483,648 to 2,147,483,647 %ld

unsigned long int 4 0 to 4,294,967,295 %lu

long long int 8 -(2^63) to (2^63)-1 %lld

unsigned long long int 8 0 to 18,446,744,073,709,551,615 %llu

signed char 1 -128 to 127 %c

unsigned char 1 0 to 255 %c

float 4 %f

double 8 %lf

long double 12 %Lf

# Enum, Struct and Union

## Enumeration (or enum)

**enum** State {Working = 1, Failed = 0};

* User defined data type.
* Assign names to integral constants,
* The names make a program easy to read and maintain.
* Size of enum varies according to range of value of enum names.

**enum** State{WORKING = 4, FAILED, FREEZED};

// sizeof(**enum** State) = 4 for 32bit PC

**enum** State{WORKING = 4294967295, FAILED, FREEZED};

// sizeof(**enum** State) = 8 for 32bit PC

1. Two enum names can have same value.

**enum** State {Working = 1, Failed = 0, Freezed = 0};

1. If we do not explicitly assign value to enum name, the compiler by default assigns values starting from 0.

**enum** day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

1. We can assign values to some name in any order. Unassigned will get value +1 of previous

**enum** day {sunday = 1, monday, tuesday = 5, wednesday, thursday = 10, friday, saturday};

1. The value assigned to enum names must be some integeral constant & with in integeral range.
2. All enum constants must be unique in their scope.

**enum** state {working, **failed**};

**enum** result {**failed**, passed};

// Compile Error: 'failed' has a previous declaration as 'state failed'

### ENUM ADVANTAGES over Macro

1. Enums follow scope rules.
2. Enum variables are automatically assigned values.

**What will be the output?**

**enum** State {WORKING = 0, FAILED, FREEZED};

**enum** State currState = 2;

**enum** State FindState() {

**return** currState;

}

**int** main() {

(FindState() == WORKING)? printf("WORKING"): printf("NOT WORKING");

**return** 0;

}

// It will work fine on C compiler as it will automatically typecast to State

// but will throw below error on C++ compiler.

// error: invalid conversion from 'int' to 'State' enum State **currState = 2**;

## Union

* User defined data type.
* All members share the same memory location
* **Size of a union** is taken according the size of largest member in union.

// Declaration of union is same as structures

**union** test{

   int x, y;

};

**int** main(){

    // A union variable t

**union** test t;

    t.x = 2; // t.y also gets value 2

    printf ("After making x = 2:\n x = %d, y = %d\n\n", t.x, t.y);

    t.y = 'A'; // t.x is also updated to A

    printf ("After making Y = 'A':\n x = %d, y = %d\n\n", t.x, t.y);

**return** 0;

}

// Output

After making x = 2:

x = 2, y = 2

After making Y = 'A':

x = 65, y = 65

# Structure Member Alignment, Padding and Data Packing

<http://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/>

# Bit Fields

http://www.geeksforgeeks.org/bit-fields-c/

* Size (in bits) of structure and union members can be specified
* The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

// A simple representation of date. **sizeof(date) = 12**

**struct** date{

**unsigned int** d;

**unsigned int** m;

**unsigned int** y;

};

// A space optimized representation of date. **sizeof(date) = 8**

**struct** date{

   // d has value between 1 and 31, so 5 bits are sufficient

**unsigned** **int** d: 5;

   // m has value between 1 and 12, so 4 bits are sufficient

**unsigned** **int** m: 4;

**unsigned** **int** y;

};

### interesting facts about bit fields

1. A special unnamed bit field of size 0 is used to force alignment on next boundary.

// A structure without forced alignment

**struct** test1{

**unsigned** **int** x: 5;

**unsigned** **int** y: 8;

};

// A structure with forced alignment

**struct** test2{

**unsigned** **int** x: 5;

**unsigned** **int: 0;**

**unsigned** **int** y: 8;

};

**int** main(){

   printf("Size of test1 is %d bytes\n", **sizeof**(struct test1));

   printf("Size of test2 is %d bytes\n", **sizeof**(struct test2));

**return** 0;

}

// Output

Size of test1 is 4 bytes

Size of test2 is 8 bytes

1. We cannot have pointers to bit field members as they may not start at a byte boundary.

**struct** test{

**unsigned** **int** x: 5;

**unsigned** **int** y: 5;

**unsigned** **int** z;

};

**int** main(){

   test t;

  // will throw an error

   printf("Address of t.x is %p", &t.x);

   // The below line works fine as z is not a bit field member

   printf("Address of t.z is %p", &t.z);

**return** 0;

}

1. It is implementation defined to assign an out-of-range value to a bit field member.

**struct** test{

**unsigned** **int** x: 2;

**unsigned** **int** y: 2;

**unsigned** **int** z: 2;

};

**int** main(){

   test t;

**t.x = 5;**

   printf("%d", t.x);

}

//Output: should be in range

Implementation-Dependent

1. Bit fields cannot be static

**struct** test1 {

**static** **unsigned** int x: 5;

};

// Error: static member 'x' cannot be a bit-field

1. Array of bit fields is not allowed.

**struct** test{

**unsigned** **int** x[10]: 5;

};

//Output

Error: bit-field 'x' has invalid type

# Little endian or big endian

# Control Statements

## Switch Case

<http://www.geeksforgeeks.org/interesting-facts-about-switch-statement-in-c/>

1. The expression used in switch must be integral type ( int, char and enum)
2. All the statements following a matching case execute until a break statement is reached.
3. The default block can be placed anywhere

**const int** i = 10;

**int** j = 10;

**int** c = 10;

**switch**(c){

**case** i: // Putting **const** before i makes the above program work

printf("Value of c = %d", c);

**break**;

**case** j: // not a "const int" expression. Not work

printf("Value of c = %d", c);

**break**;

/\*Some more cases \*/

}

## LOOPS

### For

**for**(**int** i = 0; i < 3; i++){

**printf**("loop ");

**continue**;

}

### While

**int** i = 0;

**while**(i < 3){

printf("loop "); /\* printed infinite times \*/

**continue**;

i++; /\*This statement is never executed\*/

}

/\* Good to use with condition based loop\*/

**bool** condition = **false**;

**while**(!condition){}

### DO while

* It guaranteed to execute at least one time.

**int** a = 10;

**do** {

cout << "value of a: " << a << endl;

a = a + 1;

}**while**( a < 20 );