C

27 January 2022 19:18

https://www.linkedin.com/in/imRajat/
https://github.com/im-Rajat

Following : https://www.programiz.com/c-programming

C Tutorial

Variables

```
char ch = 'a';
// some code
ch = 'l';
int number = 5;     // integer variable
number = 5.5;       // error
double number;      // error
```

Output

```
int testInteger = 5;
printf("Number = %d", testInteger);

float number1 = 13.5;
   double number2 = 12.4;

   printf("number1 = %f\n", number1);
   printf("number2 = %lf", number2);

char chr = 'a';
   printf("character = %c", chr);
```

I/O Multiple Values

```
 scanf("%d%f", &a, &b);
```

Scanning a string in c

```
Char str[20];
Scanf("%[^\n]", &str);
Or
Gets(str);
```

The sizeof operator

```
sizeof(a)

Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

C if...else Ladder

```
if (test expression1) {
  // statement(s)
}
else if(test expression2) {
  // statement(s)
```

```
}
.
.
else {
  // statement(s)
}
```

for Loop

```
for (i = 1; i < 11; ++i)
 {
   printf("%d ", i);
 }
```

C while Loop

```
while (i <= 5) {
   printf("%d\n", i);
   ++i;
 }
```

do...while loop

```
do {
   printf("Enter a number: ");
   scanf("%lf", &number);
   sum += number;
 }
 while(number != 0.0);
```

C break

The break statement ends the loop immediately when it is encountered. Its syntax is:

```
break;
```

The break statement is almost always used with if...else statement inside the loop.

C continue

The continue statement skips the current iteration of the loop and continues with the next iteration. Its syntax is:

```
continue;
```

The continue statement is almost always used with the if...else statement.

C switch Statement

```
// Program to create a simple calculator
#include <stdio.h>

int main() {
   char operation;
   double n1, n2;

   printf("Enter an operator (+, -, *, /): ");
   scanf("%c", &operation);
   printf("Enter two operands: ");
   scanf("%lf %lf",&n1, &n2);

   switch(operation)
   {
      case '+':
         printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
         break;

      case '-':
         printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
         break;
```

```
      // operator doesn't match any case constant +, -, *, /
      default:
         printf("Error! operator is not correct");
   }
```

C goto Statement

```
  for (i = 1; i <= maxInput; ++i) {
    printf("%d. Enter a number: ", i);
    scanf("%lf", &number);

    // go to jump if the user enters a negative number
    if (number < 0.0) {
      goto jump;
    }
    sum += number;
  }

jump:
  average = sum / (i - 1);
  printf("Sum = %.2f\n", sum);
  printf("Average = %.2f", average);
```

C Functions

```
#include <stdio.h>
void functionName()
{
   ... .. ...
}

int main()
{
   ... .. ...
   functionName();
   ... .. ...
}
```

C Storage Class

Every variable in C programming has two properties: *type and storage class.*

Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable.

There are 4 types of storage class:
Automatic = be default local variable
External = using a variable that declared in another file
static
Register

Register Variable

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variable s.

Static Variable

A static variable is declared by using the static keyword. For example;
static int i;
The value of a static variable persists until the end of the program.

C Arrays

```
int data[100];

int mark[5] = {19, 10, 8, 17, 9};
int mark[] = {19, 10, 8, 17, 9};
```

```
mark[2] = -1;
scanf("%d", &mark[2]);

 // taking input and storing it in an array
for(int i = 0; i < 5; ++i) {
    scanf("%d", &values[i]);
  }

// printing elements of an array
  for(int i = 0; i < 5; ++i) {
    printf("%d\n", values[i]);
  }
```

C Multidimensional Arrays

```
float x[3][4];
float y[2][4][3];

// Different ways to initialize two-dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};

int test[2][3][4] = {
   {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
   {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};

void displayNumbers(int num[][2]) {
  // code
}
```

C Pointers

Address in C : If you have a variable var in your program, &var will give you its address in the memory.

Pointers (pointer variables) are special variables that are used to store addresses rather than values.

```
int *p1;
int * p2;


int* pc, c;
c = 5;
pc = &c;
printf("%d", *pc);   // Output: 5
```

Relationship Between Arrays and Pointers

```
int x[4];
  int i;

  for(i = 0; i < 4; ++i) {
     printf("&x[%d] = %p\n", i, &x[i]);
  }

  printf("Address of array x: %p", x);

//output
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
```

&x[3] = 1450734460
Address of array x: 1450734448

There is a difference of 4 bytes between two consecutive elements of array x. It is because the size of int is 4 bytes (on ou r compiler).
Notice that, the address of &x[0] and x is the same. It's because the variable name x points to the first element of the arra y.

Basically, &x[i] is equivalent to x+i and x[i] is equivalent to *(x+i).

```
for(i = 0; i < 6; ++i) {
  // Equivalent to scanf("%d", &x[i]);
     scanf("%d", x+i);

  // Equivalent to sum += x[i]
     sum += *(x+i);
  }
```

Pass Addresses to Functions

```
swap( &n1, &n2); // Passing by Pointer
void swap(int* n1, int* n2)
{
   int temp;
   temp = *n1;
   *n1 = *n2;
   *n2 = temp;
}
```

swap(x, y); // **Passing by Reference** (Not possible in c, only in c++)

```
void swap(int& x, int& y)
{
   int z = x;
   x = y;
   y = z;
}
```

Passing Pointers to Functions

```
#include <stdio.h>

void addOne(int* ptr) {
  (*ptr)++; // adding 1 to *ptr
}

int main()
{
  int* p, i = 10;
  p = &i;
  addOne(p);

  printf("%d", *p); // 11
  return 0;
}
```

C Dynamic Memory Allocation

To allocate memory dynamically, library functions are malloc(), calloc(), realloc() and free() are used.
These functions are defined in the <stdlib.h> header file.

C malloc()

The name "malloc" stands for memory allocation.
The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.

Syntax of malloc()
ptr = (castType*) malloc(size);

Example
ptr = (float*) malloc(100 * sizeof(float));

C calloc()
The name "calloc" stands for contiguous allocation.
The malloc() function allocates memory and leaves the memory uninitialized, whereas the calloc() function allocates memory and initializes all bits to zero.

Syntax of calloc()
ptr = (castType*)calloc(n, size);

Example:
ptr = (float*) calloc(25, sizeof(float));

C free()
Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free()
free(ptr);

C realloc()
If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

Syntax of realloc()
ptr = realloc(ptr, x);
Here, ptr is reallocated with a new size x.

C Programming Strings
a string is a sequence of characters terminated with a null character \0. For example:
char c[] = "c string";

char s[5];
char c[] = "abcd";
char c[50] = "abcd";
char c[] = {'a', 'b', 'c', 'd', '\0'};
char c[5] = {'a', 'b', 'c', 'd', '\0'};

char c[100];
c = "C programming"; // Error! array type is not assignable.
Note: Use the strcpy() function to copy the string instead.

scanf() to read a string :
char name[20];
printf("Enter name: ");
scanf("%s", name);
printf("Your name is %s.", name);

Scanning a string/sentence in c
Char str[20];
Scanf("%[^\n]", &str);
Or
Gets(str);

How to read a line of text?
char name[30];

```
   printf("Enter name: ");
   fgets(name, sizeof(name), stdin);  // read string
   printf("Name: ");
   puts(name);    // display string
   return 0;
```

Note: The gets() function can also be to take input from the user. However, it is removed from the C standard.
It's because gets() allows you to input any length of characters. Hence, there might be a buffer overflow.

Strings and Pointers

```
char name[] = "Harry Potter";

  printf("%c", *name);     // Output: H
  printf("%c", *(name+1));   // Output: a
```

Commonly Used String Functions - string.h"

strlen() - calculates the length of a string
strcpy() - copies a string to another
strcmp() - compares two strings
strcat() - concatenates two strings
strlwr() converts string to lowercase
strupr() converts string to uppercase

gets() and puts()

```
   char name[30];
   printf("Enter name: ");
   gets(name);     //Function to read string from user.
   printf("Name: ");
   puts(name);    //Function to display string.
   return 0;
```

C struct : Define Structures

Syntax of struct

```
struct structureName {
  dataType member1;
  dataType member2;
  ...
};
```

For example,

```
struct Person {
  char name[50];
  int citNo;
  float salary;
};
```

Create struct Variables:

```
struct Person person1, person2, p[20];
```

Access Members of a Structure:
There are two types of operators used for accessing members of a structure.

. - Member operator
-> - Structure pointer operator

Keyword typedef

We use the typedef keyword to create an alias name for data types. It is commonly used with structures to simplify the syntax of declaring variables.

```
typedef struct Distance {
  int feet;
  float inch;
} distances;

int main() {
  distances d1, d2;
}
```

Nested Structures

```
struct complex {
  int imag;
  float real;
};

struct number {
  struct complex comp;
  int integers;
} num1, num2;
```

C structs and Pointers

to use pointers to access members of structs in C programming

C Pointers to struct

```
struct person
{
   int age;
   float weight;
};

    struct person *personPtr, person1;
    personPtr = &person1;

    printf("Enter age: ");
    scanf("%d", &personPtr->age);

    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);

    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);
```

personPtr->age is equivalent to (*personPtr).age
personPtr->weight is equivalent to (*personPtr).weight

To allocate the memory for n number of struct person, we used,
ptr = (struct person*) malloc(n * sizeof(struct person));

Passing structs to functions

```
struct student {
   char name[50];
   int age;
};

struct student s1;

   printf("Enter name: ");
```

```
  // read string input from the user until \n is entered
  // \n is discarded
  scanf("%[^\n]%*c", s1.name);

  printf("Enter age: ");
  scanf("%d", &s1.age);

  display(s1); // passing struct as an argument

void display(struct student s) {
  printf("\nDisplaying information\n");
  printf("Name: %s", s.name);
  printf("\nAge: %d", s.age);
}
```

Return struct from a function

```
struct student getInformation()
{
 struct student s1;

 printf("Enter name: ");
 scanf ("%[^\n]%*c", s1.name);

 printf("Enter age: ");
 scanf("%d", &s1.age);

 return s1;
}

struct student s;
s = getInformation();
```

Passing struct by reference

```
typedef struct Complex
{
   float real;
   float imag;
} complex;

complex c1, c2, result;
addNumbers(c1, c2, &result);

void addNumbers(complex c1, complex c2, complex *result)
{
    result->real = c1.real + c2.real;
    result->imag = c1.imag + c2.imag;
}
```

C Unions

```
union car
{
 char name[50];
 int price;
};

int main()
{
 union car car1, car2, *car3;
 return 0;
}
```

```
union car
{
  char name[50];
  int price;
} car1, car2, *car3;
```

Access members of a union
We use the . operator to access members of a union. And to access pointer variables, we use the -> operator.

the size of a union variable will always be the size of its largest element
With a union, all members share the same memory.

C enums
```
enum flag {const1, const2, ..., constN};

// Changing default values of enum constants
enum suit {
   club = 0,
   diamonds = 10,
   hearts = 20,
   spades = 3,
};

enum boolean {false, true};
enum boolean check; // declaring an enum variable


enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

int main()
{
   // creating today variable of enum week type
   enum week today;
   today = Wednesday;
   printf("Day %d",today+1);
   return 0;
}
```

C Preprocessor and Macros
C Source Code -> Preprocessor -> Compile

The C preprocessor is a macro preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be the inclusion of header file, macro expansions etc.

All preprocessing directives begin with a # symbol. For example,
```
#define PI 3.14
```

Including Header Files: #include
The #include preprocessor is used to include header files to C programs. For example,
```
#include <stdio.h>
```

Macros using #define
A macro is a fragment of code that is given a name. You can define a macro in C using the #define preprocessor directive.

Here's an example.

```
#define c 299792458  // speed of light
```

Example 1: #define preprocessor

```
#define PI 3.1415
#define circleArea(r) (PI*r*r)
```

How to use conditional?
To use conditional, #ifdef, #if, #defined, #else and #elif directives are used.

```
#ifdef Directive
#ifdef MACRO
  // conditional codes
#endif
```

C Standard Library Functions
Square root using sqrt() function

```
#include <math.h>

 root = sqrt(num);
```