Pass an Array to a C Function – 3 Easy Ways!

journaldev.com/44346/pass-array-to-c-function

August 31, 2020

What is a C Array?

Arrays in C, is a data structure that stores similar type of data values altogether at some static memory locations.

`Arrays` can be used alongside functions to deal and manipulate various data elements together.

So, let us now have a look at each of the three techniques to pass an array to a function in the upcoming section.

Technique 1: Pass an Array to a C function through call by value

In this technique, we would be passing the array to the function through call by value method.

In call by value, the value of the actual parameters are copied to the function's formal parameter list. Thus, the changes made within the function do not reflect in the actual parameter list of the calling function.

Let us have a look at the below example to get a clear idea about this technique.

```
#include <stdio.h>
void show_value( int a)
{
printf ( "%d " , a);
}
int main()
{
int a[] = {10,20,30,40,50};
printf ( "The elements of Array:\n" );
for ( int i=0; i<5; i++)
{
show_value (a[i]);
}
return 0;
}
```

So, here we have passed every element of the array to the function 'show_value' through a for loop.

Output:

```
The elements of Array:
10 20 30 40 50
```

Technique 2: Pass an Array to a C function through call by reference

In call be reference technique, we pass the address of the arguments as parameters. That is, both the actual and formal arguments refer to the same location.

Thus, any changes made inside the function are reflected in the actual parameters as well. So, let us now understand the implementation through the below example:

```
#include <stdio.h>
void show_value( int *a)
{
printf ( "%d " , *a);
}
int main()
{
int a[] = {10,20,30,40,50};
printf ( "The elements of Array:\n" );
for ( int i=0; i<5; i++)
{
show_value (&a[i]);
}
return 0;
}
```

Here, we pass the address of every element present in the array as '&array[element]'.

Output:

```
The elements of Array:
10 20 30 40 50
```

Technique 3: Pass entire array to the function directly!

In the above examples, we have passed the array to the function in an element wise fashion. That is, the elements are passed one by one.

Now, we have passed the entire array to the function directly as shown below–

```
#include <stdio.h>
void print_arr( int *arr, int size)
{
for ( int i=1; i<=size; i++)
{
printf ( "Element [%d] of array: %d \n" , i, *arr);
arr++;
}
}
int main()
{
int arr[] = {10,20,30,40,50};
print_arr(arr,5);
return 0;
}
```

Here, we have passed entire array and its size to the function.

Further, we print every element of the array using the for loop and the statement `arr++` enables us to increment the pointer and refer to the next element of the array until the condition is met.

Output:

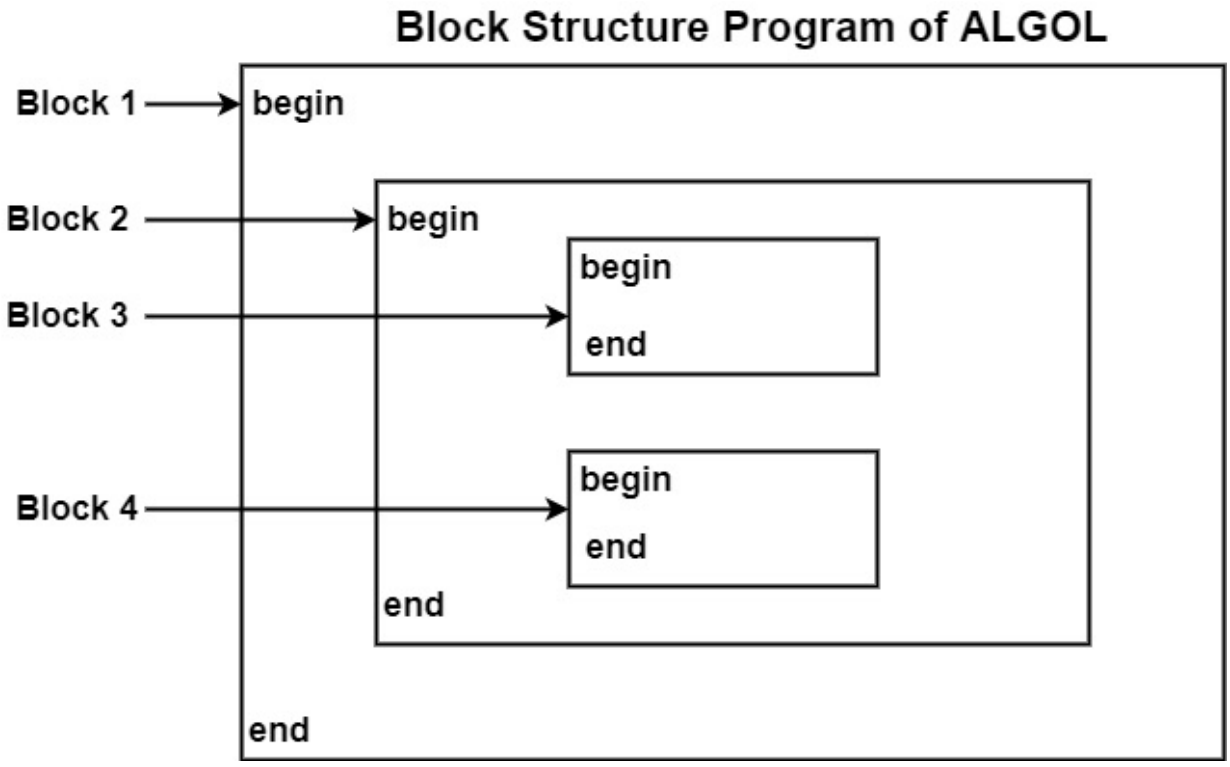
```
Element [1] of array: 10
Element [2] of array: 20
Element [3] of array: 30
Element [4] of array: 40
Element [5] of array: 50
```

What is Block Structure?

A block is a statement containing its own local data declaration. The concept of a block is originated with ALGOL. The block-structured language permits an array with adjustable length. The main feature of blocks is their bracketing structure (begin and end used in ALGOL) in which they can define their data. In 'C' language, the syntax of the block is –

```
{
   declaration statements;
}
```

where the braces limit the block. The characteristic of a block is its nesting structure. Delimiters mark the starting and end of the block. In 'C' language, the braces { } act as delimiters, while ALGOL uses begin and end. Delimiter ensures that the block is independent of one another or is nested inside the other. The nesting property is sometimes referred to as a block structure.



The most closely nested rule gives the scope of declaration –

- The scope of a declaration in Block B contains B.

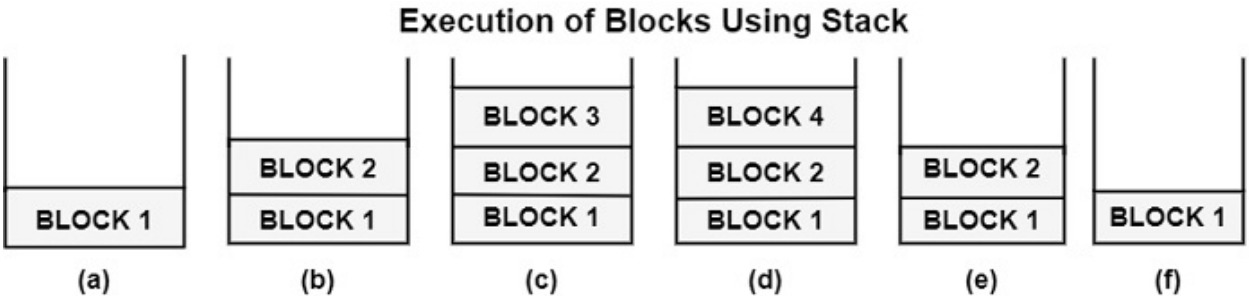
- If a name x is used in block B but is not declared in B then using a declaration of x in an enclosing block B is in the scope of a declaration of x in an enclosing block B′ such that
    - B′ has a declaration of x and
    - B′ is more closely nested around B than any other block with the declaration of x.

Consider a C program –

```
main()
{
  int a=0;
  int b=0;
{
    int b =1;
    {
      int a =2;
      printf("%d%d\n", a, b);
    }
    {
      int b = 3;
      printf("%d%d\n",a,b);
    }
    printf("%d%d\n",a,b);
  }
  printf("%d%d\n",a,b);
}
```

Block 2 Block 1 Block 0 Block 3

The execution of the above block structure program using stack can be shown in the following way –



Block structured language like ALGOL, and PL/I permit adjustable arrays, i.e., of varying length. Therefore, it cannot store irregular size arrays in between activation records. It can allocate the flexible or variable arrays at one corner of the activation record or above the fixed-size data. A pointer to these adjustable arrays will be stored in a fixed position of the activation record.

Block structured program consists of the nested procedure. Sometimes, a procedure refers to data that is not declared in it, i.e., non-local data. Non-local data will not be available in its activation record. Non-local data of one activation record will be present

in an activation record of some other procedure.

There are two methods to access Non-Local data for a procedure are as follows –

- **Static Link**– In this method, a pointer called static link is attached with each procedure which points to the topmost activation record of that procedure that physically surrounds it in the program. So, Non-local data references for any procedure can be found out by descending chain of pointers to find all statically enclosing procedures.
- **Display**– The display is an array of pointer which are maintained to speed up the access to non-local data.

C - Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a function definition in C programming language is as follows –

```
return_type function_name( parameter list ) {
   body of the function
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function –

- **Return Type** – A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two –

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts –

```
return_type function_name( parameter list );
```

For the above defined function max(), the function declaration is as follows –

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration –

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example –

Live Demo

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result –

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function –

Sr.No.	Call Type & Description
1	Call by value This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

2	Call by reference This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

C - Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

Live Demo

```
#include <stdio.h>

int main () {

  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

Live Demo

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example –

Live Demo

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of g = 10
```

Formal Parameters

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example –

Live Demo

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main () {

  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b) {

   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);

   return a + b;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows –

Data Type	Initial Default Value
int	0
char	'\0'

float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

Function call by Value in C

tutorialspoint.com/cprogramming/c_function_call_by_value.htm

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
/* function definition to swap the values */
void swap(int x, int y) {

   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */

   return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example –

Live Demo

```
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(a, b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
void swap(int x, int y) {

   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */

   return;
}
```

Let us put the above code in a single C file, compile and execute it, it will produce the following result –

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

It shows that there are no changes in the values, though they had been changed inside the function.

Advertisements

Type Conversion in C++

geeksforgeeks.org/type-conversion-in-c

October 22, 2018

1. **Implicit Type Conversion** Also known as ‘automatic type conversion’.
    - Done by the compiler on its own, without any external trigger from the user.
    - Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
    - All the data types of the variables are upgraded to the data type of the variable with largest data type.

      ```
      bool -> char -> short int -> int ->

      unsigned int -> long -> unsigned ->

      long long -> float -> double -> long double
      ```

    - It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:

```
// An example of implicit conversion

#include <iostream>
using namespace std;

int main()
{
int x = 10;  // integer x
char y =  'a' ;  // character c

// y implicitly converted to int. ASCII
// value of 'a' is 97
x = x + y;

// x is implicitly converted to float
float z = x + 1.0;

cout <<  "x = " << x << endl
<<  "y = " << y << endl
<<  "z = " << z << endl;

return 0;
}
```

Output:

```
x = 107
y = a
z = 108
```

2. **Explicit Type Conversion**: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type. In C++, it can be done by two ways:

    - **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.
      Syntax:

      ```
      (type) expression
      ```

      where *type* indicates the data type to which the final result is converted.

      Example:

      ```
      // C++ program to demonstrate
      // explicit type casting


      #include <iostream>
      using namespace std;


      int main()
      {
      double x = 1.2;


      // Explicit conversion from double to int
      int sum = ( int )x + 1;


      cout << "Sum = " << sum;


      return 0;
      }
      ```

      Output:

      ```
      Sum = 2
      ```

- **Conversion using Cast operator:** A Cast operator is an **unary operator** which forces one data type to be converted into another data type.
  C++ supports four types of casting:
  1. Static Cast
  2. Dynamic Cast
  3. Const Cast
  4. Reinterpret Cast

     Example:

```
#include <iostream>
using namespace std;
int main()
{
float f = 3.5;

// using cast operator
int b =  static_cast < int >(f);

cout << b;
}
```

     Output:

     3

Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.