**Arrays**

# Array

Stores a fixed size sequential collection of element of the same type

All arrays consist of contiguous memory location.

# Array Declaration

int arr[10]; // declaration by specifying size

int arr[] = {10, 20, 30 40}; // declaration by initializing elements

// above is same as int arr[4] = {10, 20, 30, 40};

int arr[6] = {10, 20, 30, 40}; // declaration by specifying size and initializing elements

// above is same as int arr[] = {10, 20, 30, 40, 0, 0};

# Accessing array elements

Index starts with 0 and goes till (sizeOfArray - 1)

Elements are accessed by using an integer index

arr[i] and i[arr] are same because arr[i] is \*(arr + i) and i[arr] is \*(i + arr)

# Index out of bound checking

int arr[2];

printf("%d %d", arr[3], arr[-2]);

This program compiles fine in C/C++ but may produce unexpected output when run

In C, no index out of bound checking

In C, it’s not compiler error to initialize an array with more elements than specified size

int arr[2] = {10, 20, 30, 40, 50};

In C, warning: excess elements in array initializeruage

In C++, error: too many initializers for 'int [2]'

# Properties of array in C language

* It is possible to have array of all types except void and functions.
* We can have array of void pointers and function pointers
* Array and pointer are different
* Arrays are always passed as pointer to functions
* A character array initialized with double quoted string has last element as ‘\0’
* Arrays can be allocated memory in any of the three segments: data, heap and stack
* Dynamically allocated arrays are allocated memory on heap
* Static or global arrays are allocated memory on data segment
* Local arrays are allocated memory on stack segment

Example

1. void arr[100]; // error: declaration of ‘arr’ as array of void

void \*arr[100]; // This is allowed

1. int arr[] = {10, 20, 30, 40, 50, 60};

int \* ptr = arr;

printf("size of arr: %d\n", sizeof(arr));

printf("size of ptr: %d\n", sizeof(ptr));

Output

size of arr: 24

size of ptr: 8

1. char arr[] = “geeks”; // size of arr is 6,

printf(“%d”, sizeof(arr)); // it is (‘\0’ terminated)

# Do not use sizeof() for array parameters

sizeof() should not be used to get number of elements in case when array is passed as pointer

void fun(int arr[]) {

unsigned int n = sizeof(arr) / sizeof(arr[0]);

printf(“Array size inside fun: %d\n”, n);

}

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};

unsigned int n = sizeof(arr) / sizeof(arr[0]);

printf(“Array size inside main: %d\n”, n);

fun(arr);

return 0;

}

Compilation warning: 'sizeof' on array function parameter 'arr' will return size of 'int \*'

Output

Array size inside main: 8

Array size inside fun: 2

# Variables sized array

In C, variable sized array **cannot be** initialized

In C++, variable sized array **can be** initialied

int main() {

int M = 2;

int arr[M][M] = {1};

int i, j;

for(i = 0; i < M; i++) {

for(j = 0; j < M; j++) {

printf("%d ", arr[i][j]);

}

printf("\n");

}

return 0;

}

Output

In C, error: variable-sized object may not be initialized

In C++,

1 0

0 0

# Difference between single quoted and double quoted declaration of char array

char arr[] = “geeks”; // size is 6 as it is ‘\0’ terminated

When chart array is initialized with a double quoted string and array size is not specified, compiler automatically allocates one extra space for string terminator ‘\0’

char arr[5] = “geeks”; // size is 5 as it is not terminated with ‘\0’

// compilation error in C++, works in C

In C++, error: initializer-string for array of chars is too long

char arr[] = {‘g’, ‘e’, ‘e’, ‘k’, ‘s’}; // size 5 not terminated with ‘\0’

For comma separated list of characters, compiler does not create extra space for string terminator ‘\0’

# Are array members deeply copied?

Compiler automatically performs deep copy for array members

In C++, no need to write our own copy constructor and assignment operator for array members

#include <iostream>

#include <cstring>

struct test\_arr {

char str[20];

};

int main() {

struct test\_arr st1, st2;

strcpy(st1.str, "GeeksForGeeks");

st2 = st1;

st1.str[0] = 'X';

st1.str[1] = 'Y';

std::cout << "st1.str: " << st1.str << std::endl;

std::cout << "st2.str: " << st2.str << std::endl;

return 0;

}

Output

st1.str: XYeksForGeeks

st2.str: GeeksForGeeks

// Deep copy in case of array

#include <iostream>

#include <cstring>

struct test\_arr {

char \*str;

};

int main() {

struct test\_arr st1, st2;

st1.str = new char[16];

strcpy(st1.str, "GeeksForGeeks");

st2 = st1;

st1.str[0] = 'X';

st1.str[1] = 'Y';

std::cout << "st1.str: " << st1.str << std::endl;

std::cout << "st2.str: " << st2.str << std::endl;

return 0;

}

Output

st1.str: XYeksForGeeks

st2.str: XYeksForGeeks

// Shallow copy in case of pointer and dynamically allocated memory

Initialization of a multidimensional array in C/C++

In C/C++, Initialization of a multidimensional array can have left most dimension as optional

Except left most dimension, all other dimensions must be specified

1. int a[][2] = { {1, 2}, {3, 4} }; // works
2. int a[][2][2] = { {1, 2}, {3, 4}}, // works

{{5, 6}, {7, 8}}

};

1. int a[][][2] = { {{1, 2}, {3, 4}}, // error

{{5, 6}, {7, 8}}

};

// Compilation error: declaration of ‘a’ as multidimensional array must have bounds for all dimensions except the first

# One line function for strcat() and strcmp()

In C,

void x\_strcat(char\* dest, char\* src) {

(\*dest) ? x\_strcat(++dest, src) :

( (\*dest++ = \*src++) ? x\_strcat(dest, src) : 0 );

}

* It first reaches end of the string dest using recursive call x\_strcat(++dest, src)
* Once end of the dest is reached, data is copied using (\*dest++ = \*src++)

int x\_strcmp(char\* a, char\* b){

return (\*a == \*b && \*b == ‘\0’) ? 0 :

( (\*a == \*b) ? x\_strcmp(++a, ++b) : 1 );

}

* Recursively increase a and b pointers, If \*a is not equal to \*b then 1 is return
* If we reach end of both strings at the same time then 0 is retuned

# What is the difference between char s[] and char \* s

## char[]

* char s[] = ‘geeksquiz’;
* Creates a char array which is like any other array
* We can do all array operations
* Compiler automatically adds ‘\0’ so it’s size is 10 (9 elements + ‘\0’)

#include <stdio.h>

int main() {

char s[] = "geeksquiz";

printf("%zu", sizeof(s));

s[0] = 'j';

printf("\n%s", s);

return 0;

}

Output

10

jeeksquiz

## char \*

* char \* s = “geeksquiz”;
* Creates a string literal
* String literal is stored in read only part of memory by most of compilers
* In C/C++, string literals have static storage duration any attempt at modifying them gives undefined behavior
* S is just a pointer and stores address of string literal

#include <stdio.h>

int main() {

char \* s = "geeksquiz"; // In C++, warning: ISO C++ forbids converting a string constant to 'char\*' [-Wwrite-strings]

printf("%zu", sizeof(s));

//s[0] = 'j'; // In C/C++ causes undefined behavior

printf("\n%s", s);

return 0;

}

Output

8 // size of pointer

geeksquiz

# gets() is risky to use

* It suffers from buffer overflow
* It does not do any array bound checking
* gets() keep on reading until it sees a newline character

#include <stdio.h>

int main() {

char s[4] = {0};

gets(s);

printf("%s\n", s);

return 0;

}

Output

learning cpp

learning cpp

**fgets()** makes sure that not more than MAX\_LIMIT characters are read

#include <stdio.h>

int main() {

char s[4] = {0};

fgets(s, 4, stdin);

printf("%s\n", s);

return 0;

}

Output

Learning CPP

Lea

# Write long strings in multi lines C/C++?

We can break a string at any point in the middle using two double quotes in the middle

#include <stdio.h>

int main() {

char \* str1 = "test " "one";

char \* str2 = "test " "two";

char \* str3 = "Ratnesh "

"Kumar "

"Tiwari";

printf("geeks "

"for geeks\n");

puts(str1);

puts(str2);

puts(str3);

return 0;

}

Output

geeks for geeks

test one

test two

Ratnesh Kumar Tiwari

# C function to swap strings

## Method – 1 Swap pointers

When using character pointer for strings (not arrays)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void swap1(char \*\* sptr1, char \*\* sptr2) {

fprintf(stdout, "Inside function: %s\n", \_\_func\_\_);

char \* temp = \*sptr1;

\*sptr1 = \*sptr2;

\*sptr2 = temp;

return;

}

int main() {

fprintf(stdout, "Inside function: %s\n", \_\_func\_\_);

char \* str1 = "geeks";

char \* str2 = "for geeks";

fprintf(stdout, "str1: %s str2: %s\n", str1, str2); fflush(stdout);

swap1(&str1, &str2);

fprintf(stdout, "str1: %s str2: %s\n", str1, str2);

return 0;

}

Compilation

prog.cpp: In function 'int main()':

prog.cpp:15:16: warning: ISO C++ forbids converting a string constant to 'char\*' [-Wwrite-strings]

char \* str1 = "geeks";

^

prog.cpp:16:16: warning: ISO C++ forbids converting a string constant to 'char\*' [-Wwrite-strings]

char \* str2 = "for geeks";

^

Output

Inside function: main

str1: geeks str2: for geeks

Inside function: swap1

str1: for geeks str2: geeks

## Method – 2 Swap data

When using character arrays to store strings

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void swap2(char \* str1, char \* str2) {

fprintf(stdout, "Inside function: %s\n", \_\_func\_\_);

char \* temp = (char \*)malloc( (strlen(str1)+ 1) \* sizeof(char));

memset(temp, 0, strlen(str1)+ 1);

strcpy(temp, str1);

strcpy(str1, str2);

strcpy(str2, temp);

free(temp);

return;

}

int main() {

fprintf(stdout, "Inside function: %s\n", \_\_func\_\_);

char str1[16] = "geeks";

char str2[16] = "for geeks";

fprintf(stdout, "str1: %s str2: %s\n", str1, str2); fflush(stdout);

swap2(str1, str2);

fprintf(stdout, "str1: %s str2: %s\n", str1, str2);

return 0;

}

Output

Inside function: main

str1: geeks str2: for geeks

Inside function: swap2

str1: for geeks str2: geeks

# Storage for strings in C

A string can be referred either using a character pointer or as a character array

## Strings as character arrays

Char str[4] = “GFG”; // one extra for ‘\0’

Char str[4] = {‘G’, ‘F’, ‘G’, ‘\0’}; // ‘\0’ is string termination

Strings as character arrays, are stored like other types of arrays in C

If str[] is auto variable stored in stack segment

If str[] is global or static stored in data segment

## Strings using character pointers

Can be stored in two ways

1. Read only string in a shared segment
2. Dynamically allocated in heap segment

### Read only string in a shared segment

* Directly assigned to a pointer
* Stored in read only block (data segment) i.e. shared among function

char \* str = “GFG”;

* “GFG” is stored in a shared read only location
* Pointer str is stored in a read-write memory
* You can change str to point something else but cannot change value at present str
* Used when we do not want to modify

### Dynamically allocated in heap segment

* Stored like other dynamically allocated things in C
* Can be shared among functions

char \* str;

int size = 4;

str = (char \*)malloc(sizeof(char) \* size);

\*(str + 0) = ‘G’;

\*(str + 1) = ‘F’;

\*(str + 2) = ‘G’;

\*(str + 3) = ‘\0’;

## Example-1 (Try to modify string)

#include <stdio.h>

int main() {

char \* str;

str = "GFG"; // read only part (data segment)

\*(str + 1) = 'N'; // undefined memory (modifying read only memory)

return 0;

}

Runtime Error:

Segmentation Fault (SIGSEGV)

#include <stdio.h>

int main() {

char str[] = "GFG"; // stored in stack segment

\*(str + 1) = 'N';

printf("%s\n", str);

return 0;

}

Output

GNG

#include <stdio.h>

#include <stdlib.h>

int main() {

int size = 4;

char \* str = (char \*)malloc(sizeof(char) \* size);

\*(str + 0) = 'G';

\*(str + 1) = 'F';

\*(str + 2) = 'G';

\*(str + 3) = '\0';

printf("%s\n", str);

\*(str + 1) = 'N';

printf("%s\n", str);

return 0;

}

Output

GFG

GNG

## Example – 2 (Try to return string from a function)

1. NOTE: String is stored in shared segment

Data stored remains even after return of getstring()

#include <stdio.h>

#include <stdlib.h>

char\* getString() {

char\* str = "GFG";

return str;

}

int main() {

printf("%s\n", getString());

return 0;

}

compilation warning: ISO C++ forbids converting a string constant to 'char\*'

Output

GFG

1. NOTE: String is stored in heap segment

Data returns even after return of getString()

#include <stdio.h>

#include <stdlib.h>

char\* getString() {

int size=4;

char \* str = (char \*)malloc(sizeof(char) \* size);

\*(str + 0) = 'G';

\*(1 + str) = 'F';

str[2] = 'G';

\*(str + 3) = '\0';

return str;

}

int main() {

printf("%s\n", getString());

return 0;

}

Output

GFG

1. NOTE: Prints garbage data

String is stored in stack frame

getString() and data may not be there after getString() returns

#include <stdio.h>

#include <stdlib.h>

char\* getString() {

char str[] = "GFG";

return str;

}

int main() {

printf("%s\n", getString());

return 0;

}

Compilation warning: address of local variable 'str' returned

Runtime Errors:

Segmentation Fault (SIGSEGV)

# Variable length arrays in C/C++

* We can allocate an auto array (on stack) of variable
* C supports variable sized arrays from C99 standard
* C++ standard (till C++11) does not support variable sized arrays

void fun(int n) {

int arr[n];

}

int main() {

fun(6);

}

# Difference between Array and Pointer

Pointer used for storing address of dynamically allocated arrays and for arrays which are passed as arguments to functions

#include <stdio.h>

#include <stdlib.h>

int main() {

int arr[] = {10, 20, 30, 40, 50};

int \* ptr = arr;

printf("size of arr[] %zu\n", sizeof(arr));

printf("size of ptr %zu\n", sizeof(ptr));

return 0;

}

Output

size of arr[] 20

size of ptr 8

Assigning any address to an array variable is not allowed

int arr[] = {10, 20}, x = 10;

int \* ptr = &x; // Fine

arr = &x; // Error

Following property of array make them look similar to pointer:

* Array name gives address of first element of array
* Array members are accessed using pointer arithmetic
* Array parameters are always passed as pointers, even when we use square brackets

#include <stdio.h>

#include <stdlib.h>

int fun(int ptr[]) {

int x = 10;

printf("size of ptr: %zu\n", sizeof(ptr));

ptr = &x;

printf("\*ptr: %d\n", \*ptr);

return 0;

}

int main() {

int arr[] = {10, 20, 30, 40, 50};

int \* ptr = arr;

printf("First element: %d\n", \*ptr);

printf("Third element: %d\n", arr[2]);

printf("Third element: %d\n", \*(arr + 2));

printf("Third element: %d\n", ptr[2]);

printf("Third element: %d\n", \*(ptr + 2));

fun(arr);

return 0;

}

Compilation warning: 'sizeof' on array function parameter 'ptr' will return size of 'int\*' [-Wsizeof-array-argument]

printf("size of ptr: %zu\n", sizeof(ptr));

Output

First element: 10

Third element: 30

Third element: 30

Third element: 30

Third element: 30

size of ptr: 8

\*ptr: 10

NOTE:

int array[5];

* array Pointer to the first element of the array
* &array Pointer to whole array of 5 int

# How to dynamically allocate a 2D array in C

* Using a single pointer
* Using an array of pointers
* Using pointer to a pointer
* Using a double pointer and one malloc() call for all rows

r: number of rows

c: number of columns

## Using a single pointer

Allocate memory block of size r\*c and access elements using simple pointer arithmetic

#include <stdio.h>

#include <stdlib.h>

int main() {

int r=3, c=4;

int \* arr = (int \*)malloc(r \* c \* sizeof(int));

int i=0, j=0, count=0;

for(i = 0; i < r; ++i) {

for(j = 0; j < c; ++j) {

\*(arr + ((i \* c) + j)) = ++count;

}

}

for(i = 0; i < r; ++i) {

for(j = 0; j < c; ++j) {

printf( "%2d ", \*(arr + ((i \* c) + j)) );

}

printf("\n");

}

if(NULL != arr)

free(arr); arr = NULL;

return 0;

}

Output

1 2 3 4

5 6 7 8

9 10 11 12

## Using an array of pointers

We can create an array of pointers of size r

After creating an array of pointers, we can dynamically allocate memory for every row

#include <stdio.h>

#include <stdlib.h>

int main() {

int r=3, c=4;

int i=0, j=0, count=0;

int \* arr[r];

for(i = 0; i < r; ++i) {

arr[i] = (int \*)malloc(c \* sizeof(int));

}

for(i = 0; i < r; ++i) {

for(j = 0; j < c; ++j)

arr[i][j] = ++count; // arr[i][j] = \*(\*(arr + i) + j)

}

for(i = 0; i < r; ++i) {

for(j = 0; j < c; ++j) {

printf("%2d ", arr[i][j]);

}

printf("\n");

}

for(i = 0; i < r; ++i) {

if(NULL != arr[i])

free(arr[i]); arr[i] = NULL;

}

return 0;

}

Output

1 2 3 4

5 6 7 8

9 10 11 12

## Using pointer to a pointer

We can create an array of pointers also dynamically using a double pointer

#include <stdio.h>

#include <stdlib.h>

int main() {

int r=3, c=4;

int i=0, j=0, count=0;

int \*\* arr = (int \*\*)malloc(r \* sizeof(int \*));

for(i = 0; i < r; ++i) {

arr[i] = (int \*)malloc(c \* sizeof(int));

}

for(i = 0; i < r; ++i) {

for(j = 0; j < c; ++j) {

arr[i][j] = ++count;

// arr[i][j] = \*(\*(arr + i) + j)

}

}

for(i = 0; i < r; ++i) {

for(j = 0; j < c; ++j) {

printf("%2d ", arr[i][j]);

}

printf("\n");

}

for(i = 0; i < r; ++i) {

if(NULL != arr[i])

free(arr[i]); arr[i] = NULL;

}

if(NULL != arr) {

free(arr); arr = NULL;

}

return 0;

}

Output

1 2 3 4

5 6 7 8

9 10 11 12

## Using a double pointer and one malloc() call for all rows

#include <stdio.h>

#include <stdlib.h>

int main() {

int r=3, c=4;

int i=0, j=0, count=0;

int \*\* arr = (int \*\*)malloc(r \* sizeof(int \*));

arr[0] = (int \*)malloc(r \* c \* sizeof(int));

for(i = 0; i < r; ++i) {

arr[i] = (\*arr + (c \* i));

}

for(i = 0; i < r; ++i) {

for(j = 0; j < c; ++j) {

arr[i][j] = ++count;

// arr[i][j] = \*(\*(arr + i) + j)

}

}

for(i = 0; i < r; ++i) {

for(j = 0; j < c; ++j) {

printf("%2d ", arr[i][j]);

}

printf("\n");

}

if(NULL != arr[0]) {

free(arr[0]); arr[0] = NULL;

}

if(NULL != arr) {

free(arr); arr = NULL;

}

return 0;

}

Output

1 2 3 4

5 6 7 8

9 10 11 12

# Pointer to an array

An array name is a constant pointer to the first element of the array

double \* p;

double balance[5];

p = balance;

it is legal to use array names as constant pointers and vice versa

balance – Pointer to first element of array

&balance – Pointer to whole array of 5 double elements

# Passing arrays to functions

C++/C does not allow passing an entire array as an argument to a function

You can pass a pointer to an array by specifying the array’s name without an index

3 ways to pass a single dimension array as an argument in a function

1. void myFunction(int \* param) { }
2. void myFunction(int \* param) { }
3. void myFunction(int \* param) { }

# Return array from functions in C++

C++ does not allow returning an entire array as an argument to a function

You can return a pointer to an array by specifying the array’s name without an index

int \* myFunction () {

int arr[10];

…

return arr;

}

# How to pass a 2D array as parameter

1. void myFunction(int arr[M][N]); // both dimensions are available globally
2. void myFunction(int arr[][N], int m); // 2nd dimension is available globally
3. void myFunction(int m, int n, int arr[][n]); // C99

// n must be passed before the 2D array

1. void myFunction(int \* arr, int m, int n); // using a single pointer

NOTE

Array parameters treated as pointers because of efficiency

It is inefficient to copy the array data in terms of both memory and time

# A few examples

## 1

#include <stdio.h>

int main() {

int arr[5];

printf("size of int: %zu \n", sizeof(int));

printf("%p \n", (arr + 0));

printf("%p \n", (arr + 1));

printf("%p \n", (arr + 2));

printf("%p \n", (arr + 3));

printf("%p \n", (arr + 4));

printf("%p \n", (&arr + 1));

return 0;

}

Output

size of int: 4

0x7ffe000cba40

0x7ffe000cba44

0x7ffe000cba48

0x7ffe000cba4c

0x7ffe000cba50

0x7ffe000cba54

## 2

#include <stdio.h>

int main() {

int arr[] = {1, 2, 3, 4, 5, 6};

int\* ptr = (int\*)(&arr + 1);

printf("%d\n", \*(ptr - 1));

return 0;

}

Output

6

NOTE:

&a is address of the whole array a[]

(&a + 1) gives "base address of a[] + sizeof(a) "

## 3

#include <stdio.h>

int main() {

int a[10][20][30] = {0};

a[5][2][2] = 2;

return 0;

}

Which of the following will print 2?

1. printf("%d", \*(((a + 5) + 2) + 2));
2. printf("%d", \*\*\*(((a + 5) + 2) + 2));
3. **printf("%d", \*(\*(\*(a + 5) + 2) + 2)); ✓**
4. None of these

## 4

#include <stdio.h>

int main() {

char p;

char buf[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

p = (buf + 1)[5]; // (buf + 1 + 5)

printf("%d", p);

return 0;

}

Output

7

## 5

#include <stdio.h>

int main() {

int arr[] = {1, 2, 3, 4, 5};

int (\*p)[5] = &arr;

int i = 0;

for(i = 0; i < 5; ++i) {

printf("%d ", (\*p)[i]);

}

return 0;

}

Output

1 2 3 4 5

## 6

int [] fun() { }

Compilation error: expected unqualified-id before '[' token

A function cannot have an explicit array as return type

## 7

int arr[] = {1, 2, 3, 4, 5}; // in source\_file1.c

extern int arr[]; // in source\_file2.c

In source\_file2.c, we can use sizeof() on arr to find out the actual size of arr?

* TRUE
* **FALSE** ✓

NOTE

* sizeof() operator works at compile time
* sizeof() on arr in source\_file2.c won’t work because arr in source\_file1.c is an incomplete type

## 8

int arr[50] = {0, 1, 2, [47]=47, 48, 49};

Compilation error: sorry, unimplemented: non-trivial designated initializers not supported

It will initialize arr[0], arr[1], arr[2], arr[47], arr[48] and arr[49] to 0, 1, 2, 47, 48 and 49 respectively

Remaining elements of the array would be initialized to 0

In C, Initialization of element can be done for selected elements

**In C++, It is not supported**

## 9

#include <stdio.h>

int main() {

int idx = 0;

int n = 4;

int arr1[n] = {0};

int arr2[n];

for(idx = 0; idx < n; ++idx) {

arr2[idx] = 0;

}

int arr3[4] = {1, 2, 3, 4};

printf("arr1[0]: %d, arr1[3]: %d\n", arr1[0], arr1[3]);

printf("arr2[0]: %d, arr2[3]: %d\n", arr2[0], arr2[3]);

printf("arr3[0]: %d, arr3[3]: %d\n", arr3[0], arr3[3]);

return 0;

}

* No issue with the definition of arr1 and arr2
* Initialization of arr1 is incorrect
* arr1 cannot be initialized due to its size being specified as variable

## 10

#include <stdio.h>

int size = 4;

int arr[size];

int main() {

return 0;

}

Compilation error: error: array bound is not an integer constant before ']' token

int arr[size];

Because size of array has been defined using variable outside any function

# Array as STL Container

Check document in CPP STL

# END