**Refer this:** [**https://www.geeksforgeeks.org/c-plus-plus/?ref=lbp**](https://www.geeksforgeeks.org/c-plus-plus/?ref=lbp)

**1: Undefined Behaviour**

Refer: <https://stackoverflow.com/questions/1239938/accessing-an-array-out-of-bounds-gives-no-error-why>

Also division by zero is an undefined behaviour

Welcome to every C/C++ programmer's bestest friend: Undefined Behavior.

There is a lot that is not specified by the language standard, for a variety of reasons. This is one of them.

In general, whenever you encounter undefined behavior, anything might happen. The application may crash, it may freeze, it may eject your CD-ROM drive or make demons come out of your nose. It may format your harddrive.

It may even, if you are really unlucky, appear to work correctly.

The language simply says what should happen if you access the elements within the bounds of an array. It is left undefined what happens if you go out of bounds. It might seem to work today, on your compiler, but it is not legal C or C++, and there is no guarantee that it'll still work the next time you run the program. Or that it hasn't overwritten essential data even now, and you just haven't encountered the problems, that it is going to cause — yet.

As for why there is no bounds checking, there are a couple aspects to the answer:

An array is a leftover from C. C arrays are about as primitive as you can get. Just a sequence of elements with contiguous addresses. There is no bounds checking because it is simply exposing raw memory. Implementing a robust bounds-checking mechanism would have been almost impossible in C.

In C++, bounds-checking is possible on class types. But an array is still the plain old C-compatible one. It is not a class. Further, C++ is also built on another rule which makes bounds-checking non-ideal. The C++ guiding principle is "you don't pay for what you don't use". If your code is correct, you don't need bounds-checking, and you shouldn't be forced to pay for the overhead of runtime bounds-checking.

So C++ offers the std::vector class template, which allows both. operator[] is designed to be efficient. The language standard does not require that it performs bounds checking (although it does not forbid it either). A vector also has the at() member function which is guaranteed to perform bounds-checking. So in C++, you get the best of both worlds if you use a vector. You get array-like performance without bounds-checking, and you get the ability to use bounds-checked access when you want it.

**2: What is Segmentation Fault?**

Refer: <https://stackoverflow.com/questions/2346806/what-is-a-segmentation-fault>

**Segmentation Fault occurs when**

1. **When we are accessing some memory location that has not been allocated to us**
2. **When we are writing to a read only portion of the memory etc.**

It’s a helper mechanism that keeps you from corrupting the memory and introducing hard-to-debug memory bugs.

Whenever you get a segfault you know you are doing something wrong with memory –

accessing a variable that has already been freed, writing to a read-only portion of the memory, etc.

Segmentation fault is essentially the same in most languages that let you mess with memory management,

there is no principal difference between segfaults in C and C++.

There are many ways to get a segfault, at least in the lower-level languages such as C(++).

A common way to get a segfault is to dereference a null pointer:

int \*p = NULL;

\*p = 1;

Another segfault happens when you try to write to a portion of memory that was marked as read-only:

char \*str = "Foo"; // Compiler marks the constant string as read-only

\*str = 'b'; // Which means this is illegal and results in a segfault

Dangling pointer points to a thing that does not exist anymore, like here:

char \*p = NULL;

{

    char c;

    p = &c;

}

// Now p is dangling

The pointer p dangles because it points to the character variable c that ceased to exist after the block ended.

And when you try to dereference dangling pointer (like \*p='A'), you would probably get a segfault.

**3: Dangling, void, null and wild pointers**

Refer: <https://www.geeksforgeeks.org/dangling-void-null-wild-pointers/>

**3.1 Dangling Pointers:** Dangling Pointers are those pointers which point to a memory location that has been deallocated (point to an invalid memory location). Accessing Dangling pointers is a undefined behaviour.

Dangling pointers can be produced in 3 ways:

**Deallocation of memory**

#include <iostream>

int main()

{

    int \*ptr = (int \*)malloc(sizeof(int));

    // After below free call, ptr becomes a

    // dangling pointer

    std::cout << ptr << " " << \*ptr << std::endl;

    free(ptr);

    std::cout << ptr << " " << \*ptr << std::endl;

    //why does this not give a segmentation fault as we are accessing a memory location that has been deallocated?

    //Because accessing a free'ed pointer is a undefined behaviour, The program may crash or we might still get results

    // No more a dangling pointer

    ptr = NULL;

}

**Function call:**

**Be careful while using functions which return pointers**

// The pointer pointing to local variable becomes

// dangling when local variable is not static.

#include<stdio.h>

int \*fun()

{

    // x is local variable and goes out of

    // scope after an execution of fun() is

    // over.

    int x = 5;

    return &x; //x is out of scope and so x is deleted from stack

}

// Driver Code

int main()

{

    int \*p = fun();

    fflush(stdin);

    // p points to something which is not

    // valid anymore

    printf("%d", \*p); //undefined behaviour

    return 0;

}

**Variable goes out of scope**

void main()

{

   int \*ptr;

   .....

   .....

   {

       int ch;

       ptr = &ch;

   }

   .....

   // Here ptr is dangling pointer

}

**3.2 Void pointer:** void pointer is a pointer that can contain data of any data type.

Note that: **Void pointers cannot be dereferenced and pointer arithmetic cannot be done on void pointers**

**To reference void pointers, typecasting to the respective datatypes should be done.**

**3.3 NULL Pointer**

NULL Pointer is a pointer which is pointing to nothing. In case, if we don’t have address to be assigned to a pointer, then we can simply use NULL.

int main()

{

    // Null Pointer

    int \*ptr = NULL;

    printf("The value of ptr is %p", ptr);

    return 0;

}

**NULL vs Uninitialized pointer (wild pointer) – An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment need not be a valid address for any member or object.**

**3.4 Wild Pointer**

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer.

int main()

{

    int \*p;  /\* wild pointer \*/

    int x = 10;

    // p is not a wild pointer now

    p = &x;

    return 0;

}

**Copy Constructor:** [**https://www.geeksforgeeks.org/copy-constructor-in-cpp/**](https://www.geeksforgeeks.org/copy-constructor-in-cpp/)

**Member Initializer list:** [**https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/**](https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/)

[**https://stackoverflow.com/questions/1711990/what-is-this-weird-colon-member-syntax-in-the-constructor**](https://stackoverflow.com/questions/1711990/what-is-this-weird-colon-member-syntax-in-the-constructor)