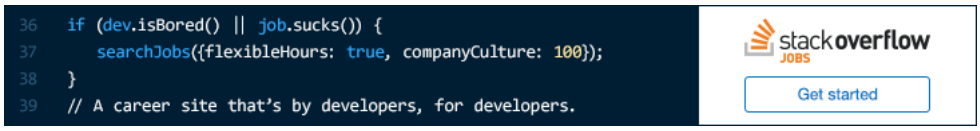
What is a segmentation fault?



What is a segmentation fault? Is it different in C and C++? How are segmentation faults and dangling pointers related?

c++ c segmentation-fault

edited Jun 1 '16 at 2:40
Jonathan Leffler
466k 66 538 860

asked Feb 27 '10 at 9:23
uppal
5,793 9 45 53

11 If that's the case, why in my case the compiler complained nothing, it all went smooth, but at run time the system throws a segmentation fault (core dump)? T_T – Jim Raynor Jan 5 '15 at 19:46

1 Just a memory dump when something goes wrong! – resultsway Apr 25 '15 at 1:18

2 @pinouchon: Funny, but when does a compiler have a thing to do with seg faults? Isn't it more the run time enviroment? – Zaibis Jul 30 '15 at 13:23

10 Answers

Segmentation fault is a specific kind of error caused by accessing memory that "does not belong to you." 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 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 the memory management, there is no principial 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 any more, like here:

```
char *p = NULL;
{
    char c;
    p = &c;
}
// Now p is dangling
```

The pointer `p` dangles because it points to 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.

edited Feb 27 '10 at 10:23
aib

answered Feb 27 '10 at 9:36
zoul

 **27.4k** 10 56 72  **69.2k** 29 192 286

81 The last example is particularly nasty, when I build: int main() { char *p = 0; { char c = 'x'; p = &c; } printf("%c\n",*p); return 0; } With either gcc or several other compilers, it 'appears' to work. No warnings on compile. No segfault. This is because the '}' out of scope, doesn't actually delete the data, just marks it as free to be used again. The code can run fine on a production system for years, you alter another part of the code, change compiler or something else and BOOOOOM! – Chris Huang-Leaver Apr 13 '10 at 9:06

13 Sorry for the bump but just a side note... none of your examples necessarily cause a segfault, in fact it's just undefined behavior ;-) – oldrinb Sep 15 '12 at 3:01

Wow, good point @Chris Huang-Leaver, thanks for lighting up this. – Kusavil May 25 '14 at 17:25

7 @oldrinb: It is impossible to write code that *necessarily* causes a segfault. Not least because there are systems out there that operate without memory protection, thus cannot tell whether a piece of memory actually "belongs to you", and thus **don't know** segfaults, only undefined behaviour... (classic AmigaOS, for example) – DevSolar May 29 '14 at 18:03

1 @ChrisHuang-Leaver, you need to understand that `c` is local, it means that it have been pushed on the stack after `{` and pop-ed out of it after `}` . the dangling pointer is just a reference to an offset which is now out of the stack. that's why modifying it in a simple program will never trigger any segfault. on the other hand it may lead to segfault in a more complex use case, where other function calls might lead the stack to grow and contain the data pointed to by the dangling pointer. writing to that data (local vars) would lead to undefined behavior (segfault &Co) – Ayman Khamouma Jan 19 '16 at 21:23

It would be worth noting that segmentation fault isn't caused by directly accessing another process memory (this is what I'm hearing sometimes), as it is simply not possible. With virtual memory every process has its own virtual address space and there is no way to access another one using any value of pointer. Exception to this can be shared libraries which are same physical address space mapped to (possibly) different virtual addresses and kernel memory which is even mapped in the same way in every process (to avoid TLB flushing on syscall, I think). And things like shmat ;) - these are what I count as 'indirect' access. One can, however, check that they are usually located long way from process code and we are usually able to access them (this is why they are there, nevertheless accessing them in a improper way will produce segmentation fault).

Still, segmentation fault can occur in case of accessing our own (process) memory in improper way (for instance trying to write to non-writable space). But the most common reason for it is the access to the part of the virtual address space that is *not mapped* to physical one at all.

And all of this with respect to virtual memory systems.

edited Mar 10 '12 at 21:58

answered Jul 3 '11 at 23:22

 konrad.kruczynski
26k 2 25 39

With shared memory/memory mapped files it is possible for someone else to mess with your memory. In WIN32 there are nasty API's like 'WriteProcessMemory' too! – paulm Feb 17 '14 at 23:46

@paulm: Yes, I know. This is what I had on mind in "And things like shmat ;) - these are what I count as 'indirect' access." – konrad.kruczynski Feb 18 '14 at 10:08

In a virtual memory operating system there's no way (normally, so please, operating system implementors, don't flame me for this) for a process to access another process virtual memory, not being some kind of memory attach system call that allows you to access. Virtual memory addresses normally mean different things depending on the process being considered. – Luis Colorado Jul 22 '16 at 12:02

A segmentation fault is caused by a request for a page that the process does not have listed in its descriptor table, or an invalid request for a page that it does have listed (e.g. a write request on a read-only page).

A dangling pointer is a pointer that may or may not point to a valid page, but does point to an "unexpected" segment of memory.

answered Feb 27 '10 at 9:27

 Ignacio Vazquez-Abrams
472k 73 846 991

5 This is true, but would it really help you if you already didn't know what a segmentation fault is? – zoul Feb 27 '10 at 9:37

1 @zoul: `homework` tag means "guide", not "give". – Ignacio Vazquez-Abrams Feb 27 '10 at 9:38

To be honest, as other posters have mentioned, Wikipedia has a very good article on this so have a look there. This type of error is very common and often called other things such as Access Violation or General Protection Fault.

They are no different in C, C++ or any other language that allows pointers. These kinds of errors are usually caused by pointers that are

1. Used before being properly initialised
2. Used after the memory they point to has been realloced or deleted.
3. Used in an indexed array where the index is outside of the array bounds. This is generally only when you're doing pointer math on traditional arrays or c-strings, not STL / Boost based collections (in C++.)

answered Feb 27 '10 at 20:35

Component 10
6,723 1 24 48

According to wikipedia:

> A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed (for example, attempting to write to a read-only location, or to overwrite part of the operating system).

answered Feb 27 '10 at 9:30

Orhan Cinar
5,762 2 22 39

While Zoul's answer explains what a segmentation fault is, I have found that these kind of bugs can be particular hard to catch, especially if you are new to low-level languages like C++ or C. Here are some of the common ways to get a segmentation fault in your program:

Improper format control string in `printf` or `scanf` statements

Format control string should have the same number of conversion specifiers (`%s` , `%d` etc.) as the `printf` or `scanf` has arguments to be printed or read. The same applies for `fprintf` and `fscanf` .

Not using `&` on the arguments to `scanf`

Function `scanf` takes as arguments the format control string and the addresses of variables in which it will place the data that it reads in. The `&` (address of) operator is used to supply the address of a variable.

Out-of-bounds array references

Make sure that you have not violated the bounds of any array you are using; i.e., you have not subscripted the array with a value less than the index of its lowest element or greater than the index of its highest element. `Valgrind` can come in handy to detect such references - you can use `valgrind` with the `--tool=exp-sgcheck` flag.

Accessing uninitialized pointers

A pointer variable must be assigned a valid address before being accessed. Make sure that you have initialized all pointers to point to a valid area of memory.

Incorrect use of the `&` (address of) and `*` (dereferencing) operators

You would need to be careful when using these, especially while passing parameters by reference/using pointers.

Shell Limits

Sometimes segmentation faults are not caused by bugs in the program but are caused instead by system memory limits being set too low. Usually it is the limit on stack size that causes this kind of problem (stack overflows). To check memory limits, use the `ulimit` command in `bash` .

Debugging using `gdb`

You can use the debugger `gdb` to view the backtrace of the `core` file dumped by your program. Whenever programs segfault, they usually dump the content of memory at the time of the crash into a `core` file (`core dumped`). Compile your program with the `-g` flag, run in `gdb` and use `bt` (backtrace).

answered Jan 10 '16 at 15:58

Madhav Datt
907 2 6 21

Segmentation fault occurs when a process (running instance of a program) is trying to access read-only memory address or memory range which is being used by other process or access the non-existent (invalid) memory address. *Dangling Reference (pointer) problem* means that

trying to access an object or variable whose contents have already been deleted from memory, e.g:

```
int *arr = new int[20];
delete arr;
cout<<arr[1];  //dangling problem occurs here
```

answered Dec 10 '13 at 22:34

Sohail xIN3N
1,575 1 18 24

2 The correct way to delete an array is delete [] arr; – Damian Mar 28 '16 at 15:48

Segmentation fault is also caused by hardware failures, in this case the RAM memories. This is the less common cause, but if you don't find an error in your code, maybe a memtest could help you.

The solution in this case, change the RAM.

edit:

Here there is a reference: Segmentation fault by hardware

edited Aug 27 '15 at 19:37

answered Jun 24 '14 at 16:59

Alejo Bernardin
187 4 12

Wikipedia's Segmentation_fault page has a very nice description about it, just pointing out the causes and reasons. Have a look into the wiki for a detailed description.

In computing, a segmentation fault (often shortened to segfault) or access violation is a fault raised by hardware with memory protection, notifying an operating system (OS) about a memory access violation.

The following are some typical causes of a segmentation fault:

- Dereferencing NULL pointers – this is special-cased by memory management hardware
- Attempting to access a nonexistent memory address (outside process's address space)
- Attempting to access memory the program does not have rights to (such as kernel structures in process context)
- Attempting to write read-only memory (such as code segment)

These in turn are often caused by programming errors that result in invalid memory access:

- Dereferencing or assigning to an uninitialized pointer (wild pointer, which points to a random memory address)
- Dereferencing or assigning to a freed pointer (dangling pointer, which points to memory that has been freed/deallocated/deleted)
- A buffer overflow.
- A stack overflow.
- Attempting to execute a program that does not compile correctly. (Some compilers will output an executable file despite the presence of compile-time errors.)

edited Apr 9 '15 at 15:36

Jamal
551 6 18 27

answered Oct 14 '14 at 10:05

fox
576 5 10

A segmentation fault or access violation occurs when a program attempts to access a memory location that is not exist, or attempts to access a memory location in a way that is not allowed.

```
/* "Array out of bounds" error
  valid indices for array foo
  are 0, 1, ... 999 */
  int foo[1000];
  for (int i = 0; i <= 1000 ; i++)
  foo[i] = i;
```

Here i[1000] not exist, so segfault occurs.

Causes of segmentation fault:

```
it arise primarily due to errors in use of pointers for virtual memory addressing,
particularly illegal access.

De-referencing NULL pointers – this is special-cased by memory management hardware.

Attempting to access a nonexistent memory address (outside process's address
space).
```

```
Attempting to access memory the program does not have rights to (such as kernel
structures in process context).

Attempting to write read-only memory (such as code segment).
```

answered Dec 8 '15 at 16:14

Mohit Rohilla
19 3

2 First of all, seg fault has nothing to do with the address does or doesn't exist. It is about you are accessing it where you are not allowed to do so. And in your special example it is even guranteed by standard that that location exists. since the standard says in array case it must be given that there is a valid address for an pointer pointg on an well aligned array within its bounds **AND 1 behind**. – Zaibis Dec 8 '15 at 16:25

it is also releated with address, if you don't have the address and if u try to access this address, also there is seg. fault. And in my example, it is only for understand point of view. – Mohit Rohilla Dec 12 '15 at 3:11

protected by Community ♦ Apr 18 '16 at 7:36

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?