7.9 — The stack and the heap

BY ALEX ON AUGUST 10TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

The memory that a program uses is typically divided into a few different areas, called segments:

- The code segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
- The bss segment (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
- The data segment (also called the initialized data segment), where initialized global and static variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The call stack, where function parameters, local variables, and other function-related information are stored.

For this lesson, we'll focus primarily on the heap and the stack, as that is where most of the interesting stuff takes place.

The heap segment

The heap segment (also known as the "free store") keeps track of memory used for dynamic memory allocation. We talked about the heap a bit already in lesson **6.9 -- Dynamic memory allocation with new and delete**, so this will be a recap.

In C++, when you use the new operator to allocate memory, this memory is allocated in the application's heap segment.

```
int *ptr = new int; // ptr is assigned 4 bytes in the heap
int *array = new int[10]; // array is assigned 40 bytes in the heap
```

The address of this memory is passed back by operator new, and can then be stored in a pointer. You do not have to worry about the mechanics behind the process of how free memory is located and allocated to the user. However, it is worth knowing that sequential memory requests may not result in sequential memory addresses being allocated!

```
int *ptr1 = new int;
int *ptr2 = new int;
// ptr1 and ptr2 may not have sequential addresses
```

When a dynamically allocated variable is deleted, the memory is "returned" to the heap and can then be reassigned as future allocation requests are received. Remember that deleting a pointer does not delete the variable, it just returns the memory at the associated address back to the operating system.

The heap has advantages and disadvantages:

- Allocating memory on the heap is comparatively slow.
- Allocated memory stays allocated until it is specifically deallocated (beware memory leaks) or the application ends (at which point the OS should clean it up).
- Dynamically allocated memory must be accessed through a pointer. Dereferencing a pointer is slower than accessing a variable directly.
- Because the heap is a big pool of memory, large arrays, structures, or classes can be allocated here.

The call stack

The **call stack** (usually referred to as "the stack") has a much more interesting role to play. The call stack keeps track of all the active functions (those that have been called but have not yet terminated) from the start of the program to the current point of execution, and handles allocation of all function parameters and local variables.

The call stack is implemented as a stack data structure. So before we can talk about how the call stack works, we need to understand what a stack data structure is.

The stack data structure

A **data structure** is a programming mechanism for organizing data so that it can be used efficiently. You've already seen several types of data structures, such as arrays and structs. Both of these data structures provide mechanisms for storing data and accessing that data in an efficient way. There are many additional data structures that are commonly used in programming, quite a few of which are implemented in the standard library, and a stack is one of those.

Consider a stack of plates in a cafeteria. Because each plate is heavy and they are stacked, you can really only do one of three things:
1) Look at the surface of the top plate
2) Take the top plate off the stack (exposing the one underneath, if it exists)
3) Put a new plate on top of the stack (hiding the one underneath, if it exists)

In computer programming, a stack is a container data structure that holds multiple variables (much like an array). However, whereas an array lets you access and modify elements in any order you wish (called **random access**), a stack is more limited. The operations that can be performed on a stack correspond to the three things mentioned above:

1) Look at the top item on the stack (usually done via a function called top(), but sometimes called peek())
2) Take the top item off of the stack (done via a function called pop())
3) Put a new item on top of the stack (done via a function called push())

A stack is a last-in, first-out (LIFO) structure. The last item pushed onto the stack will be the first item popped off. If you put a new plate on top of the stack, the first plate removed from the stack will be the plate you just pushed on last. Last on, first off. As items are pushed onto a stack, the stack grows larger -- as items are popped off, the stack grows smaller.

For example, here's a short sequence showing how pushing and popping on a stack works:

```
Stack: empty
Push 1
Stack: 1
Push 2
Stack: 1 2
Push 3
Stack: 1 2 3
Pop
Stack: 1 2
Pop
Stack: 1
```

The plate analogy is a pretty good analogy as to how the call stack works, but we can make a better analogy. Consider a bunch of mailboxes, all stacked on top of each other. Each mailbox can only hold one item, and all mailboxes start out empty. Furthermore, each mailbox is nailed to the mailbox below it, so the number of mailboxes can not be changed. If we can't change the number of mailboxes, how do we get a stack-like behavior?

First, we use a marker (like a post-it note) to keep track of where the bottom-most empty mailbox is. In the beginning, this will be the lowest mailbox (on the bottom of the stack). When we push an item onto our mailbox stack, we put it in the mailbox that is marked (which is the first empty mailbox), and move the marker up one mailbox. When we pop an item off the stack, we move the marker down one mailbox (so it's pointed at the top non-empty mailbox) and remove the item from that mailbox. Anything below the marker is considered “on the stack”. Anything at the marker or above the marker is not on the stack.

The call stack segment

The call stack segment holds the memory used for the call stack. When the application starts, the main() function is pushed on the call stack by the operating system. Then the program begins executing.

When a function call is encountered, the function is pushed onto the call stack. When the current function ends, that function is popped off the call stack. Thus, by looking at the functions pushed on the call stack, we can see all of the functions that were called to get to the current point of execution.

Our mailbox analogy above is fairly analogous to how the call stack works. The stack itself is a fixed-size chunk of memory addresses. The mailboxes are memory addresses, and the “items” we’re pushing and popping on the stack are called **stack frames**. A stack frame keeps track of all of the data associated with one function call. We’ll talk more about stack frames in a bit. The “marker” is a register (a small piece of memory in the CPU) known as the stack pointer (sometimes abbreviated “SP”). The stack pointer keeps track of where the top of the call stack currently is.

The only difference between our hypothetical mailbox stack and the call stack is that when we pop an item off the call stack, we don’t have to erase the memory (the equivalent of emptying the mailbox). We can just leave it to be overwritten by the next item pushed to that piece of memory. Because the stack pointer will be below that memory location, we know that memory location is not on the stack.

The call stack in action

Let’s examine in more detail how the call stack works. Here is the sequence of steps that takes place when a function is called:

1. The program encounters a function call.
2. A stack frame is constructed and pushed on the stack. The stack frame consists of:
    - The address of the instruction beyond the function call (called the **return address**). This is how the CPU remembers where to return to after the called function exits.
    - All function arguments.
    - Memory for any local variables.
    - Saved copies of any registers modified by the function that need to be restored when the function returns
3. The CPU jumps to the function’s start point.
4. The instructions inside of the function begin executing.

When the function terminates, the following steps happen:

1. Registers are restored from the call stack
2. The stack frame is popped off the stack. This frees the memory for all local variables and arguments.
3. The return value is handled.
4. The CPU resumes execution at the return address.

Return values can be handled in a number of different ways, depending on the computer’s architecture. Some architectures include the return value as part of the stack frame. Others use CPU registers.

Typically, it is not important to know all the details about how the call stack works. However, understanding that functions are effectively pushed on the stack when they are called and popped off when they return gives you the fundamentals needed to understand recursion, as well as some other concepts that are useful when debugging.

A technical note: on some architectures, the call stack grows away from memory address 0. On others, it grows towards memory address 0. As a consequence, newly pushed stack frames may have a higher or a lower memory address than the previous ones.

A quick and dirty call stack example

Consider the following simple application:

```
int foo(int x)
{
    // b
    return x;
} // foo is popped off the call stack here

int main()
{
    // a
    foo(5); // foo is pushed on the call stack here
    // c

    return 0;
}
```

The call stack looks like the following at the labeled points:

a:

```
main()
```

b:

```
foo() (including parameter x)
main()
```

c:

```
main()
```

Stack overflow

The stack has a limited size, and consequently can only hold a limited amount of information. On Windows, the default stack size is 1MB. On some unix machines, it can be as large as 8MB. If the program tries to put too much information on the stack, stack overflow will result. **Stack overflow** happens when all the memory in the stack has been allocated -- in that case, further allocations begin overflowing into other sections of memory.

Stack overflow is generally the result of allocating too many variables on the stack, and/or making too many nested function calls (where function A calls function B calls function C calls function D etc...) On modern operating systems, overflowing the stack will generally cause your OS to issue an access violation and terminate the program.

Here is an example program that will likely cause a stack overflow. You can run it on your system and watch it crash:

```
#include <iostream>

int main()
{
    int stack[10000000];
    std::cout << "hi";
    return 0;
}
```

This program tries to allocate a huge (likely 40MB) array on the stack. Because the stack is not large enough to handle this array, the array allocation overflows into portions of memory the program is not allowed to use.

On Windows (Visual Studio), this program produces the result:

```
HelloWorld.exe (process 15916) exited with code -1073741571.
```

-1073741571 is c0000005 in hex, which is the Windows OS code for an access violation. Note that “hi” is never printed because the program is terminated prior to that point.

Here’s another program that will cause a stack overflow for a different reason:

```
void foo()
{
    foo();
}

int main()
{
    foo();

    return 0;
}
```

In the above program, a stack frame is pushed on the stack every time function foo() is called. Since foo() calls itself infinitely, eventually the stack will run out of memory and cause an overflow.

The stack has advantages and disadvantages:

- Allocating memory on the stack is comparatively fast.
- Memory allocated on the stack stays in scope as long as it is on the stack. It is destroyed when it is popped off the stack.
- All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
- Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes passing by value or creating local variables of large arrays or other memory-intensive structures.

7.10 -- std::vector capacity and stack behavior

Index

7.8 -- Function Pointers

C++ TUTORIAL | PRINT THIS POST

188 comments to 7.9 — The stack and the heap

« Older Comments 1 2 3

Nirbhay
September 17, 2019 at 1:39 am · Reply

Hello!

What is the meaning of the following line of code:

```
int *array = new int[10];
```

array is a pointer which points to an address in the heap that contains an int with value = 10?

OR

array is a pointer which is assigned 10 int (40 bytes with one int being 4 bytes) in the heap?

Thanks :)

nascardriver
September 17, 2019 at 2:30 am · Reply

\`array\` is a pointer which points to the first of 10 consecutive \`int\`s. Those \`int\`s are most likely on the heap, but the standard doesn't require it.

Nirbhay
September 17, 2019 at 4:03 am · Reply

How can array point to first 10 consecutive ints? I mean it can let's say store the address of the first int but what about the other 9 consecutive ints?

nascardriver
September 17, 2019 at 4:18 am · Reply

You can calculate their positions using pointer arithmetic, or access them using array syntax

```
array[3]
```

The pointer doesn't know whether it's pointing to a single \`int\` or to an array of \`int\`. It's up to the coder to know what their pointers are pointing to.

Nirbhay
September 17, 2019 at 4:35 am · Reply

Alright.. in the below Video,

https://www.youtube.com/watch?v=CSVRA4_xOkw

from 8:00 to 9:00 , it is shown if,

int* p = new int(10);

then pointer p has the address of a memory location in a heap where an int with initial value of 10 is created.

So the explanations here on Learncpp and the video seem opposite to each other. Or am I missing something?

Please help.

Thanks :)

nascardriver
September 17, 2019 at 4:54 am · Reply

```
new int(10); // Parentheses (Direct initialization)
new int[10]; // Square brackets (Array)

new int{ 10 }; // Curly brackets (Brace initialization)
new int[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // Array with brace in
itialization
```

Nirbhay
September 17, 2019 at 4:57 am · Reply

Noted! Thanks!

new int[10]; // Square brackets (Array)

All 10 ints will have garbage values here right?

nascardriver
September 17, 2019 at 5:00 am · Reply

If the array type is a fundamental type (int, float, etc.), yes.
If not, the elements will be default-initialized (You'll learn about classes later).
If the array is made of `std::string` for example, all elements are empty strings.
If you want to initialize the array, you can use empty curly braces as always.

```
new int[10]{}; // All 0
```

Nirbhay
September 17, 2019 at 5:01 am · Reply

Thanks :)

Samira Ferdi
September 5, 2019 at 5:38 pm · Reply

Hi, Alex and Nascardriver!

Is it true that stack memory allocation has automatic duration and scope-dependent and heap memory allocation has dynamic duration and scope-independent?

nascardriver
September 6, 2019 at 12:38 am · Reply

C++ doesn't put any requirements on the underlying memory model.
But yes, dynamic allocations usually use the heap and automatic allocations use the stack.

Samira Ferdi
September 4, 2019 at 6:00 pm · Reply

Hi, Alex (and Nascardriver too)

I don't undestand your mailboxes analogy. I can't imagine and just don't get it. Your plate analogy is perfect by the way. I'm sorry Alex, in my place there is no any mailboxes.

HD
December 24, 2019 at 6:43 am · Reply

Hello Alex (or Nascardriver),
I am also unable to understand the mailboxes analogy, but I want to understand it. Can you please elaborate that in detail so that I can understand.
Thanks in advance.
And a huge thank you for this amazing tutorial also.

Alex
January 1, 2020 at 8:18 pm · Reply

Try drawing a picture of a bunch of mailboxes stacked on top of each other, and an arrow pointed at the lowest mailbox. For each item pushed on the stack, put the item in the mailbox being pointed to and move the arrow up. For each item being popped, move the arrow down and remove the item from the mailbox that is now being pointed to. That's all it is.

Kcnkcn
December 25, 2019 at 5:53 pm · Reply

The plate analogy is great for understanding how to use a stack, but it does not quite accurately represent what the stack looks like. I suppose this isn't important for learning c++ (apart from stack overflows) but it's important for learning how a computer works.
(note: I am a student, so I may not be 100% accurate in my explanation)

For computers, you only get a certain amount of memory. As Alex says, "On Windows, the default stack size is 1MB. On some unix machines, it can be as large as 8MB."
You cannot create more memory (like the old joke "download more RAM"), nor can you destroy memory (or else you wouldn't be able to get it back).

In the plate analogy, you can keep adding/pushing more plates without ever hitting a cap. Also, when you take/pop plates, the volume of the plate stack takes up decreases. Since memory in the stack has to stay constant (not right either because it shares space with the heap, but close enough), the plate analogy fails to take this into consideration.

The mailbox analogy is like the plate analogy except that the memory is already laid out. Each mailbox represents space to put a chunk of data into and the entire stack of mailbox represents all of the memory given to you. You cannot remove mailboxes nor can you add more because you would anger the authorities. Instead of putting a plate on top of the plate stack, you place it inside the bottom-most available mailbox.
But where is the bottom-most available mailbox? You can't look at the top of the stack (while the plate analogy incorrectly allows you to) because all of the unused mailboxes are blocking your view. So you have to keep track of where the bottom-most available mailbox is by yourself (the push() and pop() functions keep track for you), i.e. through a marker or post-it. Whenever you push or pop something, you know which mailbox is the bottom-most available mailbox because of the post-it. When you push,

remember to move the post-it up 1 mailbox for the new bottom-most available mailbox. Same for popping except you move 1 down.

The reason I say "available" instead of "empty" is because when you pop some data, you can't empty it out because it needs to have some 0s and 1s in it. You could set the memory to all 0s, but I think it would take too long and maybe a few other restraints. So instead, we just move the post-it down 1 mailbox and just overwrite the data in the bottom-most available mailbox if we have to re-use it.

Piyush
April 27, 2019 at 8:55 am · Reply

"Because the stack is not large enough to handle this array, the array allocation overflows into portions of memory the program is not allowed to use. Consequently, the program crashes."

So, does the memory overflowed by the array allocation is restored to the previous contents or the overflow does'nt really changes the content of the memory.

nascardriver
April 27, 2019 at 9:19 am · Reply

The code in question doesn't change the memory, it merely reserves it, no data has been overridden.
Data is never automatically restored, and there's no reason to do so, your program crashed anyway.

Barne
March 18, 2019 at 2:06 pm · Reply

Hey Alex, in the foo example option c seems to be lacking code tags around main().

Great lesson, really interesting reading about the low level code and functionality!

Alex
March 19, 2019 at 8:10 pm · Reply

Thanks! The formatting has been fixed.

Sonu
February 24, 2019 at 9:20 am · Reply

Hi Alex,

In advantages and disadvantages of stacks, following line is mentioned:
"All memory allocated on the stack is known at compile time."

As per my understanding, the stack frames are created/destroyed when a function is called/returned.
So how memory allocated on the stack is known at compile time.??

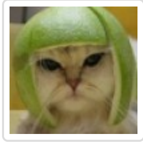
Alex
February 25, 2019 at 2:09 pm · Reply

The compiler is responsible for laying out all of the stack frames at compile time (with local variables translated into an memory address offset from the start of the stack frame).

It doesn't necessarily know (or care) which stack frame is actively loaded into memory at a given point.

hassan magaji
February 5, 2019 at 9:33 am · Reply

hi Alex,
i am missing something under the bss segment(unitialized data segment)?
my point here is that "aren't zero initialid g_variables really initialized?"
i think they should be stored in the data segment where initialize g_variables are stored.



Alex
February 7, 2019 at 6:14 am · Reply

If I'm wrong, then **Wikipedia** is wrong too.

Roey
December 26, 2018 at 2:57 am · Reply

The first line of this page is written "The memory a program", should be "The memory *of* a program".

Akshay Gajbe
September 7, 2018 at 11:18 pm · Reply

```
#include <iostream>

class MyClass
{
private:
    int variable1;
    int variable2;

public:
    int SetVariable()
    {
        //TO DO
    }

};

int main()
{

    std::cout<<"Size of Class: "<<sizeof(MyClass);
    return 0;
}
```

outPut:
Size of Class: 8

1. here it shows size of the class as 8, it means memory allocated for class variable (variable1 & variable2),but what about the class member function? where it gets allocate memory for member function?

2. But if I made member function virtual

```
virtual int SetVariable()
{
}
```

then it shows
size of Class: 16

Why is that so?

nascardriver
September 8, 2018 at 7:34 am · Reply

Hi Akshay!

> what about the class member function?
Functions aren't members of class instances, because the function is the same across all instances. It is created once, somewhere in memory, and every occurrence of the function is replaces with the address the function is located at.

> But if I made member function virtual
Classes with functions store a pointer to their virtual function table. You're compiling in 64 bit mode, so that pointer is 8 bytes in size, 8 (variables) + 8 (vtable pointer) = 16. You can add more virtual functions without increasing the class size, because each instance only stores a pointer to the vtable, not the vtable itself. More about this in chapter 12.

Akshay Gajbe
September 13, 2018 at 10:12 am · Reply

Thanks @nascardriver, my confusion got cleared now.

Akshay Gajbe
September 7, 2018 at 5:24 am · Reply

Hi Alex,
here it is mentioned that
"The call stack is a fixed-size chunk of memory addresses."
but in https://codingfox.com/10-8-memory-stack-segment-call-stack/
it mentioned that 'The stack grows towards the lower part of memory that is towards heap. In other hand heap grows towards higher part of memory that is towards stack.'
that means call stack size is not fixed.

nascardriver
September 7, 2018 at 5:30 am · Reply

The maximum stack size is fixed, but the stack itself can grow and shrink.

Peter Baum
May 11, 2018 at 4:02 pm · Reply

Readers might assume that a stack grows from lower memory addresses to higher if they follow the plate analogy. On the standard PC x86 architecture the stack grows toward address zero. That could lead to confusion if looking at actual addresses. It might be something worth mentioning.

Alex
May 19, 2018 at 8:52 am · Reply

Added a technical note about this to the article. Thanks for mentioning!

Mahmud
February 21, 2018 at 4:25 pm · Reply

Alex,

1. Since dynamically created arrays (or any variable type) stay on the heap and not on the stack, does it mean that, their scopes are global? Say, we have a main() function that calls another function createArray() that creates an array dynamically. When the program returns to the main, will it be able to refer to the array created without createArray() having to return the dynamically created array?

2. In case returning the array to the main() from createArray() is mandatory, is the return type (for the createArray() function), when returning lets say a 2 dimensional array with integer elements, int** ?

nascardriver
February 22, 2018 at 5:50 am · Reply

Hi Mahmud!

1:
The array still exists after @createArray has finished, however there is reliable way for accessing it anymore, since you don't know where it is.

2:
Yes.

```
int **createArray(size_t x, size_t y)
{
  int **arr{ new int *[x] };

  for (size_t s{ 0 }; s < x; ++s)
  {
    arr[s] = new int[y];
  }

  return arr;
}

int main()
{
  int **arr{ createArray(2, 3) };

  // We need to delete the sub-arrays before deleting the array of arrays.
  for (size_t s{ 0 }; s < 2; ++s)
  {
    delete[] arr[s];
  }

  delete[] arr;

  return 0;
}
```

Bonus (1):

```
#include <iostream>
```

```

void createArray(size_t x, size_t y)
{
  int **arr{ new int *[x] };

  for (size_t s{ 0 }; s < x; ++s)
  {
    arr[s] = new int[y];
  }

  std::cout << "A " << x << "x" << y << " Array has been allocated at " << std::hex
<< arr << std::endl;
}

int main()
{
  createArray(2, 3);

  uintptr_t uiAddress{ 0 };

  std::cout << "Where's the array at?" << std::endl;
  std::cin >> std::hex >> uiAddress;

  int **arr{ reinterpret_cast<int **>(uiAddress) };

  // Use @arr here

  // Deletion omitted

  return 0;
}
```

Mahmud
February 22, 2018 at 12:18 pm · Reply

@nascardriver
Thanks a lot for the clarification!

Clapfish
December 9, 2018 at 2:41 pm · Reply

Hi nascardriver!

I found this code (Bonus (1)) interesting so I started playing with it.

In the '// Use @arr here' section I added the following:

```
arr[1][1] = 365;
std::cout << arr[1][1];
```

This outputs '16d' (hex) rather than '365'.

However, what we seem to want from this code is an accessible/usable two dimensional 'normal' int array (unless I'm mistaken).

I'm guessing the issue is being caused by the iostream and use of std::hex somehow?

How could it be resolved?

nascardriver
December 10, 2018 at 9:39 am · Reply

Once you set a flag in a stream, it will stay set.
You can switch back to decimal using @std::dec

```
std::cout << std::dec;
```

Everything after this line will be decimal

Alex
February 26, 2018 at 11:01 am · Reply

1) Nascardriver's answer is good. But it sounds like you're confusing the concepts of scope and duration. Duration has to do with how long something lives for. Scope has to do with where you can access something. So for a dynamically allocated array, it has dynamic duration (meaning it lives until you explicitly destroy it). Dynamically allocated memory doesn't have "scope", since you never access it directly. Instead, you access it through pointers. Those pointers have scope.

PiotrLenarczyk
January 16, 2018 at 6:19 am · Reply

```
//is it stack - friendly code?
#include <vector>
using std::vector;

void foo( void );
int main( void )
{
   foo();
   return 0;
};

void foo( void )
{
   const uint32_t N = 1E9;
   vector < float > vec;
   vec.resize( N );
};
```

Alex
January 18, 2018 at 5:47 pm · Reply

Yes, because vectors allocate their elements on the heap. However, if this were a std::array, that would not be the case.

An easy way to tell is this: if you take the sizeof() your variable, that's how many bytes on the stack that variable is taking. Even though your vector has a lot of elements, the sizeof(vec) should be relatively small.

Imre B.
November 18, 2017 at 12:54 pm · Reply

Hi Alex!

I have a question. Does making a new block creates a new stack aswell? Or variables only delete because they're out of scope?

Alex
November 20, 2017 at 11:18 am · Reply

No, blocks inside a function don't create an additional stack frame. Compilers have some leeway in terms of how they handle the variables inside functions, but most allocate room for variables defined in the function (regardless of nested blocks) on the stack at the start of the function. The compiler can enforce the proper scoping rules.

Liam
November 14, 2017 at 8:51 am · Reply

Two questions I've had for a while now, but haven't been able to find answers to.

1. Are the stack and the heap (and other types of memory) physically different? Ie. Could you open up you machine and point to the stack as being distinct from the heap? Or are they just different software allocations within a homogeneous RAM chip?

2. Why is LIFO an efficient way to order memory? From a naive perspective, it seems like it would be most inefficient, ie. if I want to access something on the bottom, I have to remove every item from the top.

Alex
November 14, 2017 at 11:34 pm · Reply

1) It depends on what you mean by distinct. But generally, yes, they're distinct portions of memory. The stack is fixed in size, and the heap has a variable size. A given memory address could belong to one or the other, but not both simultaneously. How these are mapped to actual RAM is determined by the OS (virtual memory makes this complicated).

2) LIFO is useful because it's very simple and fast to add or remove the top element, and it ensures all items in the stack are contiguous. If you need to access something on the bottom, you probably shouldn't use a stack in the first place.

nikos-13
July 12, 2017 at 8:34 am · Reply

Could you describe me what a "register" is, or give me an example?

Alex
July 12, 2017 at 1:05 pm · Reply

A register is a small piece of memory that is part of the CPU. Many CPUs load data from main memory into registers, operate on the registers, and then output the result of the register back to main memory.

Curiosity
June 11, 2017 at 6:23 am · Reply

1) Saved copies of any registers modified by the function that need to be restored when the function returns.

2) Return values can be handled in a number of different ways, depending on the computer's architecture. Some architectures include the return value as part of the stack frame. Others use CPU registers.

I didn't understand these 2 points.
Can you explain me these? Reply Please.

Alex
June 11, 2017 at 3:32 pm · Reply

This would require explaining a lot about computer architectures, which isn't really the point of these tutorials. Just note this as an interesting aside and move on.

Curiosity
June 11, 2017 at 7:12 pm · Reply

Should i read on net about registers?

Alex
June 12, 2017 at 1:44 pm · Reply

If you're super interested in computer architecture and the intimate details of how all this stuff works, sure. But if your primary interest is learning C++ then I'd bookmark it to return to later.

James
May 30, 2017 at 3:35 am · Reply

Hello Alex,
When is the best time to use vector instead of array and vice versa?

Alex
May 30, 2017 at 2:23 pm · Reply

Use std::array when you know the array's length at compile-time and it won't change. Use std::vector if you don't know the array's length until runtime, or the length needs to change.

Amr
May 21, 2017 at 11:01 pm · Reply

Hello alex ,
First i want to thank you for this tutorial, but i deployed a multithreaded program which when allocate all memory on the stack use arrays instead of vectors for example

```
func x(some parameters){
vector  <double >a;
a.resize(4)
vector <double> e;
a.resize(4)
vector <double> c;
c.resize(4)
vector  <bool> d;
d.resize(4)
vector <bool> x;
x.resize(4)
for( int i =0 ; i <100000 ; i++ )
for (int j =0 ; j <100000; j++ )
{
double inphase[4];
double quadrature[4];
bool status [4];
bool hidden [4];
here some reading and writng in those arrays and vectors

}
}
```

program executes well but one transforming the vecyors to arrays program becomes 10x slower how is that and when switching array with vectors and vectors wth arrays program executes fast again , i am using open mp and visual studio 2010 by the way the parallization is for this function each thread call this function seperately.

Alex
May 22, 2017 at 1:22 pm · Reply

I'm not sure. In general, std::array should be faster than std::vector (at the cost of being less flexible). You must be accruing some kind of inefficiency with std::array or stack allocation, perhaps because multiple threads don't share a stack.

amr
May 23, 2017 at 12:55 am · Reply

and that exactly what i need to that threads don't share the stack what is the drawback of threads not sharing same stack ?

Alex
May 23, 2017 at 3:27 pm · Reply

I'm not the right person to ask. I haven't done multithreaded programming in a long time, and am not all that familiar with the pros and cons of it.

amr
May 23, 2017 at 11:19 pm · Reply

Thanks lot for your concern alex

AlexR
May 11, 2017 at 3:30 pm · Reply

Hello!

That was an interesting read. I had a random idea popped in my head that might sound dumb. I understand the difference between the stack and the heap (I think) but I wonder is it possible to use a pointer to point to another stack instead of grab memory from the heap? Instead of calling a function and pushing onto the main stack, I was thinking maybe the pointer can allocate the function call to another stack and then you could push additional functions onto that.

I guess the only thing that might be wrong there is that the OS will terminate the program for trying to access memory it wasn't given. Is that right?

I'm not sure if I explained that clearly but in my mind I imagine something like:

```
STACK 1 - - - - - - - - - - - - - - - STACK 2
4 (points to main in stack 2)           2
3                                       1
2                                      main
1
main
```

Alex
May 12, 2017 at 1:17 pm · Reply

A program is only set up with one stack, and the compiler is set up to automatically use it. So when you call a function, it goes on the main stack. I'm not aware of any way to reroute to a second stack.

Eno
March 16, 2017 at 8:00 am · Reply

Dear Alex,
I have a module and I want to save it in a persistent memory in order not to be lost or crashed when I call it again starting from the point where I left it, and I found the only way to do that is to transfer it to a disk after saving it in a file using mmap command, but the problem is I found this command can not be executed under Windows OS because it's included in <sys/mman.h> which is one of the Linux header files. So is there an alternative command can be executed under Windows platform using vc++.

Regards.

Alex
March 16, 2017 at 11:22 am · Reply

I can't really speak to this, as it's OS specific and not in my core area of knowledge. Google search or Stack Overflow is probably your best bet here.

Eno
March 1, 2017 at 12:44 am · Reply

Dear Alex,

I couldn't because each project has its header files and main function! so how can I merge these two projects together in one project. Actually, I'm looking for steps to do that.

Regards.

Alex
March 1, 2017 at 5:06 pm · Reply

The header files you should just be able to copy into your interface project. The main function I'm not sure about.

Satish Gandham

February 27, 2017 at 9:52 pm · Reply

Hi Alex,
Thanks for the very detailed guide. Very easy to understand for someone who's not a CS major.

« Older Comments 1 2 3