Can all functions in a thread have access to dynamically allocated memory (heap) even without passing pointer or is it local to a function?

Asked 15 years, 9 months ago Modified 3 years, 6 months ago Viewed 6k times

1

I have a very basic question and need help. I am trying to understand what is the scope of a dynamically allocated memory (on heap).

```
#include <stdio.h>
#include <malloc.h>

//-----Struct def-------
struct node {
        int x;
        int y;
};

//------GLOBAL DATA------

//-----FUNC DEFINITION----
void funct(){
  t->x = 5; //**can I access 't' allocated on heap in main over here ?**
  t->y = 6; //**can I access 't' allocated on heap in main over here ?**
  printf ("int x = %d\n", t->x);
  printf ("int y = %d\n", t->y);
  return;
}

//-----MAIN FUNCTION------
int main(void){
      struct node * t = NULL;// and what difference will it make if I define
                                //it outside main() instead- as a global
pointer variable
          t = (struct node *) malloc (sizeof(struct node));
      t->x = 7;
      t->y = 12;
      printf ("int x = %d\n", t->x);
      printf ("int y = %d\n", t->y);

    funct(); // FUNCTION CALLED**
    return 0;
}
```

Here, can I access structure `t` in `funct()` even though the memory is allocated in `main()` without passing argument (pointer to `t` to `function funct`) - since heap is common to a thread? What difference will it make if I define `struct node * t = NULL` outside of `main()` as a global variable and is there anything wrong with it?

c memory-management dynamic heap-memory

Share
Improve this question
Follow

edited Jun 13, 2021 at 21:29

trincot
348k ● 35 ● 269 ● 319

asked Mar 28, 2009 at 11:27

a g

Thank you all. Thanks wekempf for making clear the difference between two concepts. Blank summed it well, so will mark it as correct answer. – a g Mar 28, 2009 at 11:48

10 Answers

Sorted by: Highest score (default)

8

When you use malloc(), the memory returned by that can be accessed anywhere in your code, assuming that you can see the variable which has the pointer returned by malloc().

So in your code, if t was global, it would be visible in main and in funct(), and yes, you could use it in both.

As it is, as previous answers have mentioned, funct() has no idea what t is, because the declaration and definition of t are in main; t is out of scope in funct. The memory you've allocated onto t *would* be useable in funct, *if* funct knew what t was.

Share
Improve this answer
Follow

edited Mar 28, 2009 at 19:29

bk1e
24.3k ● 6 ● 57 ● 65

answered Mar 28, 2009 at 11:40

user82238

3

No, you will not be able to access t like that because even though in main you made t point to something on the heap, which is globally accessible, the variable t itself is a pointer which is local to main (and located on the stack).

Share Improve this answer Follow

answered Mar 28, 2009 at 11:36

Ferruccio

101k ● 38 ● 229 ● 303

The fact that the pointer is allocated on the stack isn't relevant to the discussion. Stack variables can easily be accessed by other threads as well. – wekempf Mar 28, 2009 at 11:39

Was this a multi-threading question? I didn't get a sense that it was. I thought it was purely about scope. – Ferruccio Mar 28, 2009 at 12:10

3

To access a variable the function needs to know the address of this variable - the value of "t" pointer. So you must pass it into the function, otherwise the function will have no means to know what to access.

You could as well declare a global variable pinter "t" and use it from any function but again explicitly.

Share Improve this answer Follow

answered Mar 28, 2009 at 11:36

sharptooth

170k ● 108 ● 535 ● 1k

2

The memory allocated can be accessed, but that's not your problem. The variable 't' is *visible* only within *main*, because that's where you declared it. Those are two different concepts that you'll need to understand. The storage for data is not the same thing as the variable that refers to it.

Share Improve this answer Follow

answered Mar 28, 2009 at 11:37

wekempf

2,778 ● 16 ● 16

1

In your code 't' is a local variable of main() - it cannot be accessed from anywhere else. This has nothing to do with threading. Please correct your code.

Share Improve this answer Follow

answered Mar 28, 2009 at 11:36

anon

Just change your function to accept the pointer to that memory.

```
/* changing the variable name to make it clear that p is scoped to funct() */
/* when it's passed, just as t is local to main() */
void funct(struct node *p){
p->x = 5;
p->y = 6;
printf ("int x = %d\n", p->x);
printf ("int y = %d\n", p->y);
return;
}
```

...

```
funct(t);

/* now free the memory you allocated */
free(t);
return 0;
```

This is the whole point of allocating memory on the heap - automatic (stack) variables are limited to the scope of the block you declare them in. Heap memory is available anywhere - you just need to pass a pointer so that other functions know where to find what you stored.

Share Improve this answer Follow

answered Mar 28, 2009 at 11:45

Mitch Flax

622 ● 1 ● 6 ● 14

1

No, you would need to make t global, or pass the pointer to funct(), I'd recommend doing the second. Right now, t is in the scope of main() only.

You might also consider adding a mutex to the structure, if you plan on having several threads operating on t at once. If you do that, also consider mlock() to ensure the structure is not paged out, or locks get slow.

Share

Improve this answer

Follow

edited Mar 28, 2009 at 12:52

answered Mar 28, 2009 at 11:40

user50049

The accepted answer is a little misleading.

> When you use malloc(), the memory returned by that can be accessed anywhere in your code, assuming that you can see the variable which has the pointer returned by malloc().

This is equally true of any memory allocated in C or C++, whether heap, stack or static. Just put & in front of a local variable, and you now have the key that allows any other part of the program to get at the storage for that variable, and use it just as effectively as if they could see the variable name itself.

Local variable names are only visible inside the scope they are declared in, but their storage is accessible from anywhere, in any thread, assuming the address has been taken and passed around.

It dangerous, and yet frequently necessary, to take the address of local variables in C/C++, and so it is unfortunately impossible to use those languages effectively without understanding this.

UPDATE

@kevinf says:

> Using the address of a local within the scope of the local is fine, but your suggestion is NOT memory safe

It's important to distinguish between *scope* and *lifetime*. Scope refers to identifiers (e.g. variable names), and it is the set of locations in your code where the identifier can be mentioned. It's a compile-time concept.

This is a separate concept from memory safety, or lifetime of an object in a storage location. This is a runtime concept. For example:

```
void g(int *p)
{
    (*p)++;
}

void f()
{
    int n = 1;
    g(&n);
}
```

Here, the identifier `n` is only in scope within `f`. It names a storage location that exists while `f` is running. We get the address `&n` of that storage location and pass it to `g`. Note that `g` cannot use the name `n` directly. `n` is not in scope here! Yet the storage

location is perfectly usable *in this example* because `f` is still unfinished (waiting for `g` to complete).

Most real C and C++ programs do this a lot, and it is allowed by the standard. It is neither generally safe nor generally unsafe. It is sometimes safe and sometimes not. This is the core challenge with C and C++. Undefined behaviour is not always detectable from local examination of a function; you have to know how it is being used in a wider context.

More recent changes to the language (in g++ use the `-std=c++14` option to compile this) introduce other ways of exploring this grey area:

```
auto f()
{
    int n = 1;

    auto g = [&]
    {
        ++n;
        cout << "Incremented to " << n << endl;
    };

    g();

    return g;
}

void h()
{
    auto g = f();

    cout << "Returned into h" << endl;

    g();
}
```

Inside `f`, `n` is in scope. And `g` holds an instance of a lambda, the body of which is also part of the scope of `n`. And the call to `g` inside `f` is perfectly fine. But in `f` when we store the lambda in another variable called `g`, the subsequent call to it is not allowed! We have, without ever using `&n`, implicitly captured a storage location that is no longer available to us, because its lifetime was limited to the duration of our call to `f`.

By the way, when I say not allowed, it will compile. But on my system it prints:

```
Incremented to 2
Returned into h
Incremented to 167772162
```

Which clearly demonstrates undefined behaviour.

Share

Improve this answer

Follow

edited May 4, 2017 at 8:34

answered Mar 28, 2009 at 11:59

Daniel Earwicker

117k • 38 • 208 • 286

Using the address of a local within the scope of the local is fine, but your suggestion is NOT memory safe stackoverflow.com/questions/6441218/… – Kevin May 4, 2017 at 3:33

1 @kevinf - you are possibly confusing scope with lifetime. I've updated my answer with examples to illustrate the difference (over 8 years later!) – Daniel Earwicker May 4, 2017 at 8:28

Example you gave is safe "int n = 1" did not go out of *lifetime*, so its present on stack. re: SO link provided, OP returning the address of a local from a function, which MAY work, but stack will quickly override memory storage. Statements "...heap, stack or static. Just put & in front of a local variable ... allows any other part of the program to get at the storage for that variable ... as if they could see the variable name itself." and "Local variable ... their storage is accessible from anywhere ... assuming the address has been taken and passed around." must have note about *lifetime* – Kevin May 4, 2017 at 14:20

That is the purpose of the update (and on the original answer the use of the word "dangerous"). Is it still unclear? – Daniel Earwicker May 4, 2017 at 14:23

0

as t is a pointer to the structure defined in the function main. you cannot have an access to it in funct(). Make t global so that it can be accessed in the funct() or pass the pointer as a argument to funct(). making it global is not the best choice. In a multithreaded environment muiltiple threads may try to access it so u need to have some kind of locking mechanism to avoid access at the same time. adding the locking it functionality may lead to other overhead so the best choice should be to pass it as the argument.

Share Improve this answer Follow

answered Mar 29, 2009 at 7:44

ruchira

0

t is a pointer variable local to the main(),but the content inside 't' exists through out the program until/unless we explicitly free it.The life time of the address that t holds is through out the program as its dynamically allocated in the heap.So you can make use of that memory address anywhere with in the program but not the variable local variable 't'.Since t is a local variable,it has only that block scope and can be used only with in the main()>So the best way to make use of t as well as its content is to declare t and then dynamically alloacate memory in the global scope.

In short variable 't' is local but the value in 't' is in heap.Only way to acess the heap memory is via 't'.So declare t in the global space,so we can acess the heap memory

anywhere in the program via 't' .Else how you will acess the heap memory,when t goes out of scope

Share Improve this answer Follow

answered Feb 8, 2013 at 8:27

Subi Suresh

301 ● 1 ● 4 ● 10