C Memory Management

Asked 16 years, 4 months ago Modified 3 years, 2 months ago

Viewed 56k times

108

I've always heard that in C you have to really watch how you manage memory. And I'm still beginning to learn C, but thus far, I have not had to do any memory managing related activities at all.. I always imagined having to release variables and do all sorts of ugly things. But this doesn't seem to be the case.

Can someone show me (with code examples) an example of when you would have to do some "memory management" ?

c memory

Share

Improve this question

Follow

edited Sep 2, 2008 at 1:54

Chris Hanson

55k ● 8 ● 74 ● 104

asked Aug 24, 2008 at 6:50

The.Anti.9

44.6k ● 49 ● 126 ● 162

Good place to learn G4G – EsmaeelE Dec 18, 2017 at 22:46

12 Answers Sorted by: Highest score (default)

251

There are two places where variables can be put in memory. When you create a variable like this:

```
int  a;
char c;
char d[16];
```

The variables are created in the "**stack**". Stack variables are automatically freed when they go out of scope (that is, when the code can't reach them anymore). You might hear them called "automatic" variables, but that has fallen out of fashion.

Many beginner examples will use only stack variables.

The stack is nice because it's automatic, but it also has two drawbacks: (1) The compiler needs to know in advance how big the variables are, and (2) the stack space is somewhat limited. For example: in Windows, under default settings for the Microsoft linker, the stack is set to 1 MB, and not all of it is available for your variables.

If you don't know at compile time how big your array is, or if you need a big array or struct, you need "plan B".

Plan B is called the "**heap**". You can usually create variables as big as the Operating System will let you, but you have to do it yourself. Earlier postings showed you one way you can do it, although there are other ways:

```
int size;
// ...
// Set size to some value, based on information availa
// ...
char *p = (char *)malloc(size);
```

(Note that variables in the heap are not manipulated directly, but via pointers)

Once you create a heap variable, the problem is that the compiler can't tell when you're done with it, so you lose the automatic releasing. That's where the "manual releasing" you were referring to comes in. Your code is now responsible to decide when the variable is not needed anymore, and release it so the memory can be taken for other purposes. For the case above, with:

```
free(p);
```

What makes this second option "nasty business" is that it's not always easy to know when the variable is not needed anymore. Forgetting to release a variable when you don't need it will cause your program to consume more memory that it needs to. This situation is called a "leak". The "leaked" memory cannot be used for anything until your program ends and the OS recovers all of its resources. Even nastier problems are possible if you release a heap variable by mistake *before* you are actually done with it.

In C and C++, you are responsible to clean up your heap variables like shown above. However, there are languages and environments such as Java and .NET languages like C# that use a different approach, where the heap gets cleaned up on its own. This second method, called "garbage collection", is much easier on the developer but you pay a penalty in overhead and performance. It's a balance.

(I have glossed over many details to give a simpler, but hopefully more leveled answer)

Share Improve this answer Follow

edited Oct 11, 2021 at 7:42

answered Aug 24, 2008 at 8:21

Euro Micelli

33.9k ● 8 ● 53 ● 72

3 If you want to put something on the stack but don't know how big it is at compile time, alloca() can enlarge the stack frame to make room. There is no freea(), the entire stack frame is popped when the function returns. Using alloca() for large allocations is fraught with peril. – DGentry Sep 7, 2008 at 2:17

1 Maybe you could add one or two sentences about the memory location of global variables – Michael Käfer Jan 18, 2017 at 22:02

Here's an example. Suppose you have a strdup() function that duplicates a string:

```
char *strdup(char *src)
{
    char * dest;
    dest = malloc(strlen(src) + 1);
    if (dest == NULL)
        abort();
    strcpy(dest, src);
    return dest;
}
```

And you call it like this:

```
main()
{
    char *s;
    s = strdup("hello");
    printf("%s\n", s);
    s = strdup("world");
    printf("%s\n", s);
}
```

You can see that the program works, but you have allocated memory (via malloc) without freeing it up. You have lost your pointer to the first memory block when you called strdup the second time.

This is no big deal for this small amount of memory, but consider the case:

```
for (i = 0; i < 1000000000; ++i)  /* billion times */
    s = strdup("hello world");    /* 11 bytes */
```

You have now used up 11 gig of memory (possibly more, depending on your memory manager) and if you have not crashed your process is probably running pretty slowly.

To fix, you need to call free() for everything that is obtained with malloc() after you finish using it:

```
s = strdup("hello");
free(s);  /* now not leaking memory! */
s = strdup("world");
...
```

Hope this example helps!

Share Improve this answer

Follow

answered Aug 24, 2008 at 7:38

Mark Harrison

304k ● 131 ● 350 ● 489

I like this answer better. But I got a little side question. I'd expect something like this to be solved with libraries, isn't there a library that would mimic closely the basic datatypes and add memory freeing features to them so that when variables get used they also get freed automatically? – Lorenzo Aug 17, 2011 at 12:44

None that are part of the standard. If you go into C++ you get strings and containers that do automatic memory management. – Mark Harrison Aug 18, 2011 at 0:44

I see, so there are some 3rd party libs? Could you please name them? – Lorenzo Aug 18, 2011 at 8:48

10

You have to do "memory management" when you want to use memory on the heap rather than the stack. If you don't know how large to make an array until runtime, then you have to use the heap. For example, you might want to store something in a string, but don't know how large its contents will be until the program is run. In that case you'd write something like this:

```
 char *string = malloc(stringlength); // stringlength allocate

 // Do something with the string...

 free(string); // Free the allocated memory
```

Share Improve this answer

Follow

answered Aug 24, 2008 at 6:57

Paige Ruten

176k ● 37 ● 182 ● 199

10

I think the most concise way to answer the question in to consider the role of the pointer in C. The pointer is a lightweight yet powerful mechanism that gives you immense freedom at the cost of immense capacity to shoot yourself in the foot.

In C the responsibility of ensuring your pointers point to memory you own is yours and yours alone. This requires an organized and disciplined approach, unless you forsake pointers, which makes it hard to write effective C.

The posted answers to date concentrate on automatic (stack) and heap variable allocations. Using stack allocation does make for automatically managed and convenient memory, but in some circumstances (large buffers, recursive algorithms) it can lead to the horrendous problem of stack overflow. Knowing exactly how much memory you can allocate on the stack is very dependent on the system. In some embedded scenarios a few dozen bytes might be your limit, in some desktop scenarios you can safely use megabytes.

Heap allocation is less inherent to the language. It is basically a set of library calls that grants you ownership of a block of memory of given size until you are ready to return ('free') it. It sounds simple, but is associated with untold programmer grief. The problems are simple (freeing the same memory twice, or not at all [memory leaks], not allocating enough memory [buffer overflow], etc) but difficult to avoid and debug. A hightly disciplined approach is absolutely mandatory in practive but of course the language doesn't actually mandate it.

I'd like to mention another type of memory allocation that's been ignored by other posts. It's possible to statically allocate variables by declaring them outside any function. I think in general this type of allocation gets a bad rap because it's used by global variables. However there's nothing that says the only way to use memory allocated this way is as an undisciplined global variable in a mess of spaghetti code. The static allocation method can be used simply to avoid some of the pitfalls of the

heap and automatic allocation methods. Some C programmers are surprised to learn that large and sophisticated C embedded and games programs have been constructed with no use of heap allocation at all.

Share Improve this answer

Follow

answered Sep 2, 2008 at 3:39

Bill Forster

6,287 ● 3 ● 29 ● 30

5

There are some great answers here about how to allocate and free memory, and in my opinion the more challenging side of using C is ensuring that the only memory you use is memory you've allocated - if this isn't done correctly what you end up with is the cousin of this site - a buffer overflow - and you may be overwriting memory that's being used by another application, with very unpredictable results.

An example:

```
int main() {
    char* myString = (char*)malloc(5*sizeof(char));
    myString = "abcd";
}
```

At this point you've allocated 5 bytes for myString and filled it with "abcd\0" (strings end in a null - \0). If your string allocation was

```
myString = "abcde";
```

You would be assigning "abcde" in the 5 bytes you've had allocated to your program, and the trailing null character would be put at the end of this - a part of memory that hasn't been allocated for your use and could be free, but could equally be being used by another application - This is the critical part of memory management, where a mistake will have unpredictable (and sometimes unrepeatable) consequences.

Share Improve this answer

Follow

answered Aug 24, 2008 at 7:58

Chris B-C

1,434 ● 14 ● 11

1 Here you allocate 5 bytes. Loose it by assigning a pointer. Any attempt to free this pointer leads to undefined behavior. Note C-Strings do not overload the = operator there is no copy. – Loki Astari Sep 23, 2008 at 0:15

Though, it really depends on the malloc you are using. Many malloc operators align to 8 bytes. So if this malloc is using a header/footer system, malloc would reserve 5 + 4*2 (4 bytes for both header and footer). That would be 13 bytes, and malloc would just give you an extra 3 bytes for alignment. I'm not saying that it's a good idea to use this, because it will only systems whose malloc works like this, but It's at least important to know why doing something wrong might work. – kodai Dec 15, 2009 at 1:11

Loki: I've edited the answer to use `strcpy()` instead of `=`; I assume that was Chris B-C's intention. – echristopherson Aug 31, 2013 at 19:08

1 I believe in modern platforms hardware memory protection prevents userspace processes from overwriting other processes' address spaces; you'd get a segmentation fault

instead. But that's not part of C per se. – echristopherson Aug 31, 2013 at 19:23

4

A thing to remember is to *always* initialize your pointers to NULL, since an uninitialized pointer may contain a pseudorandom valid memory address which can make pointer errors go ahead silently. By enforcing a pointer to be initialized with NULL, you can always catch if you are using this pointer without initializing it. The reason is that operating systems "wire" the virtual address 0x00000000 to general protection exceptions to trap null pointer usage.

Share Improve this answer

Follow

answered Mar 29, 2009 at 23:00

Hernán

4,587 ● 4 ● 33 ● 48

2

Also you might want to use dynamic memory allocation when you need to define a huge array, say int[10000]. You can't just put it in stack because then, hm... you'll get a stack overflow.

Another good example would be an implementation of a data structure, say linked list or binary tree. I don't have a sample code to paste here but you can google it easily.

Share Improve this answer

Follow

answered Aug 24, 2008 at 7:50

Serge

7,694 ● 5 ● 41 ● 46

2

(I'm writing because I feel the answers so far aren't quite on the mark.)

The reason you have to memory management worth mentioning is when you have a problem / solution that requires you to create complex structures. (If your programs crash if you allocate to much space on the stack at once, that's a bug.) Typically, the first data structure you'll need to learn is some kind of list. Here's a single linked one, off the top of my head:

```
typedef struct listelem { struct listelem *next; void

listelem * create(void * data)
{
   listelem *p = calloc(1, sizeof(listelem));
   if(p) p->data = data;
   return p;
}
```

```
listelem * delete(listelem * p)
{
   listelem next = p->next;
   free(p);
   return next;
}

void deleteall(listelem * p)
{
  while(p) p = delete(p);
}

void foreach(listelem * p, void (*fun)(void *data) )
{
  for( ; p != NULL; p = p->next) fun(p->data);
}

listelem * merge(listelem *p, listelem *q)
{
  while(p != NULL && p->next != NULL) p = p->next;
  if(p) {
    p->next = q;
    return p;
  } else
    return q;
}
```

Naturally, you'd like a few other functions, but basically, this is what you need memory management for. I should point out that there are a number tricks that are possible with "manual" memory management, e.g.,

- Using the fact that malloc is guaranteed (by the language standard) to return a pointer divisible by 4,
- allocating extra space for some sinister purpose of your own,
- creating memory pools..

Get a good debugger... Good luck!

Share Improve this answer

Follow

answered Aug 24, 2008 at 14:13

Anders Eurenius

4,226 ● 2 ● 25 ● 20

> Learning data-structures is the next key step in understanding memory management. Learning the algorithms to appropriately run these structures will show you the appropriate methods to overcome these hurtles. This is why you will find Data-structures and Algorithms taught in the same courses. – aj.toulan Nov 4, 2013 at 17:47

0

@Euro Micelli

One negative to add is that pointers to the stack are no longer valid when the function returns, so you cannot return a pointer to a stack variable from a function. This is a common error and a major reason why you can't get by with just stack variables. If your function needs to return a pointer, then you have to malloc and deal with memory management.

Share Improve this answer

Follow

edited May 23, 2017 at 12:34

Community Bot

1 ● 1

answered Aug 25, 2008 at 16:50

Jonathan Branam

744 ● 5 ● 10

0

> @Ted Percival:
> ...you don't need to cast malloc()'s return value.

You are correct, of course. I believe that has always been true, although I don't have a copy of K&R to check.

I don't like a lot of the implicit conversions in C, so I tend to use casts to make "magic" more visible. Sometimes it helps readability, sometimes it doesn't, and sometimes it causes a silent bug to be caught by the compiler. Still, I don't have a strong opinion about this, one way or another.

> This is especially likely if your compiler understands C++-style comments.

Yeah... you caught me there. I spend a lot more time in C++ than C. Thanks for noticing that.

Share Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 • 1

answered Aug 25, 2008 at 16:09

Euro Micelli

33.9k • 8 • 53 • 72

@echristopherson, thanks. You are right - but please note that this Q/A was from August 2008, before Stack Overflow was even in public Beta. Back then, we were still figuring out

how the site should work. The format of this Question/Answer should not necessarily be viewed as a model for how to use SO. Thanks! – Euro Micelli Aug 31, 2013 at 23:50

Ah, thanks for pointing that out -- I didn't realize that aspect of the site was still in flux then. – echristopherson Sep 1, 2013 at 2:53

0

In C, you actually have two different choices. One, you can let the system manage the memory for you. Alternatively, you can do that by yourself. Generally, you would want to stick to the former as long as possible. However, auto-managed memory in C is extremely limited and you will need to manually manage the memory in many cases, such as:

a. You want the variable to outlive the functions, and you don't want to have global variable. ex:

```
struct pair{
   int val;
   struct pair *next;
}

struct pair* new_pair(int val){
   struct pair* np = malloc(sizeof(struct pair));
   np->val = val;
   np->next = NULL;
   return np;
}
```

b. you want to have dynamically allocated memory. Most common example is array without fixed length:

```
int *my_special_array;
my_special_array = malloc(sizeof(int) *
number_of_element);
for(i=0; i

c. You want to do something REALLY dirty. For
example, I would want a struct to represent many
kind of data and I don't like union (union looks
soooo messy):



struct data{
  int data_type;
  long data_in_mem;
};

struct animal{/*something*/};
struct person{/*some other thing*/};

struct animal* read_animal();
struct person* read_person();

/*In main*/
struct data sample;
sampe.data_type = input_type;
switch(input_type){
 case DATA_PERSON:
   sample.data_in_mem = read_person();
   break;
 case DATA_ANIMAL:
   sample.data_in_mem = read_animal();
 default:
   printf("Oh hoh! I warn you, that again and I
will seg fault your OS");
}
```

See, a long value is enough to hold ANYTHING. Just remember to free it, or you WILL regret. This is among my favorite tricks to have fun in C :D.

However, generally, you would want to stay away from your favorite tricks (T___T). You WILL break your OS, sooner or later, if you use them too often. As long as you don't use *alloc and free, it is safe to say that you are still virgin, and that the code still looks nice.

Share Improve this answer

Follow

answered Aug 27, 2008 at 12:08

magice

447 ● 3 ● 4

"See, a long value is enough to hold ANYTHING" - :/ what are you talking about, on most systems a long value is 4 bytes, exactly the same as an int. The only reason it fits the pointers in here is because the size of long happens to be the same as the pointer size. You should really be using void*, though. – Score_Under Sep 26, 2012 at 16:30

-3

Sure. If you create an object that exists outside of the scope you use it in. Here is a contrived example (bear in mind my syntax will be off; my C is rusty, but this example will still illustrate the concept):

```
class MyClass
{
   SomeOtherClass *myObject;

   public MyClass()
   {
      //The object is created when the class is constr
      myObject = (SomeOtherClass*)malloc(sizeof(myObje
   }

   public ~MyClass()
   {
```

```
        //The class is destructed
        //If you don't free the object here, you leak me
        free(myObject);
    }

    public void SomeMemberFunction()
    {
        //Some use of the object
        myObject->SomeOperation();
    }


};
```

In this example, I'm using an object of type SomeOtherClass during the lifetime of MyClass. The SomeOtherClass object is used in several functions, so I've dynamically allocated the memory: the SomeOtherClass object is created when MyClass is created, used several times over the life of the object, and then freed once MyClass is freed.

Obviously if this were real code, there would be no reason (aside from possibly stack memory consumption) to create myObject in this way, but this type of object creation/destruction becomes useful when you have a lot of objects, and want to finely control when they are created and destroyed (so that your application doesn't suck up 1GB of RAM for its entire lifetime, for example), and in a Windowed environment, this is pretty much mandatory, as objects that you create (buttons, say), need to exist well outside of any particular function's (or even class') scope.

Share Improve this answer

Follow

answered Aug 24, 2008 at 7:00

TheSmurf

15.6k ● 3 ● 41 ● 48

2 Heh, yeah, that's C++ isn't it? Amazing that it took five months for anyone to call me on it. – TheSmurf Jan 21, 2009 at 5:59