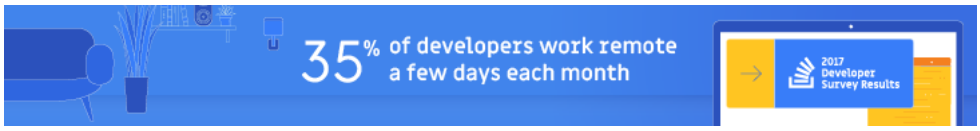
In what cases do I use malloc vs new?



I see in C++ there are multiple ways to allocate and free data and I understand that when you call `malloc` you should call `free` and when you use the `new` operator you should pair with `delete` and it is a mistake to mix the two (e.g. Calling `free()` on something that was created with the `new` operator), but I'm not clear on when I should use `malloc` / `free` and when I should use `new` / `delete` in my real world programs.

If you're a C++ expert, please let me know any rules of thumb or conventions you follow in this regard.

c++ memory-management malloc new-operator

edited Aug 10 '14 at 14:11
Unihedron
8,270 10 40 61

asked Oct 8 '08 at 19:47
Ralph Burgess

19 I'd just like to add a reminder that you cannot mix the two styles - that is, you cannot use new to create an object and then call free() on it, nor attempt to delete a block allocated by malloc(). Probably obvious to say it, but nonetheless... – nsayer Oct 8 '08 at 19:53

18 Good answers, all I have to add (that I haven't seen) is that new/delete calls the constructor/destructor for you, malloc/free does not. Just a difference worth mentioning. – Bill K Oct 8 '08 at 20:07

30 check edit history -- craziness going on with this question stackoverflow.com/revisions/184537/list – Jeff Atwood ♦ Oct 8 '08 at 20:18

27 Wait, so was he editing his own question to include the offensive stuff? Maybe a new badge for multiple personality disorder? – swilliams Oct 8 '08 at 21:23

10 Ralph probably didn't get any answers at that second, so naturally he got bored and decided to attract more reader by replacing the question with erotic words. He was banned from the website and found himself lost in his own thoughts. Ralph moved to Tibet and lived among the monks, and there he started writing erotic posts as a professional...still reading? – ArmenB Jan 7 '15 at 22:41

16 Answers

Unless you are forced to use C, you should **never use malloc**. Always use new. If you need a big chunk of data just do something like:

```
char *pBuffer = new char[1024];
```

Be careful though this is not correct:

```
//This is incorrect - may delete only one element, may corrupt the heap, or
worse...
delete pBuffer;
```

Instead you should do this when deleting an array of data:

```
//This deletes all items in the array
delete[] pBuffer;
```

The **new keyword** is the C++ way of doing it, and it will ensure that your type will have their **constructor called**. The new keyword is also more **type safe** whereas malloc is not typesafe at all.

The only way I could think that would be beneficial to use **malloc** would be if you needed to **change the size of your buffer** of data. The new keyword does not have an analogous way

like realloc. The realloc function might be able to extend the size of a chunk of memory for you more efficiently.

It is worth mentioning that you cannot mix new/free and malloc/delete.

Note, some answers in this question are invalid.

```
int* p_scalar = new int(5);//Does not create 5 elements, but initializes to 5
int* p_array = new int[5];//Creates 5 elements
```

edited Apr 30 '09 at 17:15

answered Oct 8 '08 at 19:48

Brian R. Bondy

209k 87 492 576

21 If you do not use the correct delete **the result is undefined**. It's incorrect. The fact that it might get part of the thing right or work sometimes is just blind luck. – Michael Burr Oct 8 '08 at 23:31

6 @KPexEA: Even if some compilers might fix your mistakes, it's still wrong to make them in the first place :) Always use delete[] where appropriate. – korona Oct 9 '08 at 8:33

37 "Unless you are forced to use C, you should never use malloc. Always use new." Why? What is the win here? For objects we need construction, but for memory blocks, you clearly document two ways to make coding mistakes (the more easily caught () vs [] in new and the less easily caught mismatched array vs scaler new and delete). What is the motivation for using new/delete for blocks of raw memory? – Ben Supnik Feb 11 '10 at 20:35

3 @DeadMG: If one is creating an array for use by an asynchronous API function, wouldn't `new[]` be much safer than `std::vector` ? If one uses `new[]` , the only way the pointer would become invalid would be via explicit `delete` , whereas the memory allocated for an `std::vector` could invalidated when the vector is resized or leaves scope. (Note that when using `new[]` one would have to allow for the possibility that one may not be able to call `delete` if the async method is still pending; if it may be necessary to abandon an async operation, one may have to arrange for delete via callback). – supercat Feb 18 '13 at 16:30

19 Downvote for "Never use X!" without explaining why. – immibis Feb 20 '14 at 22:44

The short answer is: don't use `malloc` for C++ without a really good reason for doing so. `malloc` has a number of deficiencies when used with C++, which `new` was defined to overcome.

Deficiencies fixed by new for C++ code

1. `malloc` is not typesafe in any meaningful way. In C++ you are required to cast the return from `void*` . This potentially introduces a lot of problems:

```
#include <stdlib.h>

struct foo {
  double d[5];
};

int main() {
  foo *f1 = malloc(1); // error, no cast
  foo *f2 = static_cast<foo*>(malloc(sizeof(foo)));
  foo *f3 = static_cast<foo*>(malloc(1)); // No error, bad
}
```

2. It's worse than that though. If the type in question is POD (plain old data) then you can semi-sensibly use `malloc` to allocate memory for it, as `f2` does in the first example.

   It's not so obvious though if a type is POD. The fact that it's possible for a given type to change from POD to non-POD with no resulting compiler error and potentially very hard to debug problems is a significant factor. For example if someone (possibly another programmer, during maintenance, much later on were to make a change that caused `foo` to no longer be POD then no obvious error would appear at compile time as you'd hope, e.g.:

```
struct foo {
  double d[5];
  virtual ~foo() { }
};
```

   would make the `malloc` of `f2` also become bad, without any obvious diagnostics. The example here is trivial, but it's possible to accidentally introduce non-PODness much further away (e.g. in a base class, by adding a non-POD member). If you have C++11/boost you can use `is_pod` to check that this assumption is correct and produce an error if it's not:

```
#include <type_traits>
#include <stdlib.h>

foo *safe_foo_malloc() {
  static_assert(std::is_pod<foo>::value, "foo must be POD");
```

```
  return static_cast<foo*>(malloc(sizeof(foo)));
}
```

Although boost is unable to determine if a type is POD without C++11 or some other compiler extensions.

3. `malloc` returns `NULL` if allocation fails. `new` will throw `std::bad_alloc`. The behaviour of later using a `NULL` pointer is undefined. An exception has clean semantics when it is thrown and it is thrown from the source of the error. Wrapping `malloc` with an appropriate test at every call seems tedious and error prone. (You only have to forget once to undo all that good work). An exception can be allowed to propagate to a level where a caller is able to sensibly process it, where as `NULL` is much harder to pass back meaningfully. We could extend our `safe_foo_malloc` function to throw an exception or exit the program or call some handler:

```
#include <type_traits>
#include <stdlib.h>

void my_malloc_failed_handler();

foo *safe_foo_malloc() {
  static_assert(std::is_pod<foo>::value, "foo must be POD");
  foo *mem = static_cast<foo*>(malloc(sizeof(foo)));
  if (!mem) {
     my_malloc_failed_handler();
     // or throw ...
  }
  return mem;
}
```

4. Fundamentally `malloc` is a C feature and `new` is a C++ feature. As a result `malloc` does not play nicely with constructors, it only looks at allocating a chunk of bytes. We could extend our `safe_foo_malloc` further to use placement `new`:

```
#include <stdlib.h>
#include <new>

void my_malloc_failed_handler();

foo *safe_foo_malloc() {
  void *mem = malloc(sizeof(foo));
  if (!mem) {
     my_malloc_failed_handler();
     // or throw ...
  }
  return new (mem)foo();
}
```

5. Our `safe_foo_malloc` function isn't very generic - ideally we'd want something that can handle any type, not just `foo`. We can achieve this with templates and variadic templates for non-default constructors:

```
#include <functional>
#include <new>
#include <stdlib.h>

void my_malloc_failed_handler();

template <typename T>
struct alloc {
  template <typename ...Args>
  static T *safe_malloc(Args&&... args) {
    void *mem = malloc(sizeof(T));
    if (!mem) {
       my_malloc_failed_handler();
       // or throw ...
    }
    return new (mem)T(std::forward(args)...);
  }
};
```

Now though in fixing all the issues we identified so far we've practically reinvented the default `new` operator. If you're going to use `malloc` and placement `new` then you might as well just use `new` to begin with!

edited Jan 13 '12 at 18:24

answered Nov 1 '11 at 17:04

Flexo ♦

57.9k 17 112 178

13 It's too bad C++ made `struct` and `class` mean basically the same thing; I wonder if there would have been any problems with having `struct` be reserved for PODs and possibly having all `class` types be presumed to be non-PODs. Any types defined by code which predated the invention of C++ would necessarily be PODs, so I wouldn't think backward compatibility would be an issue there. Are there advantages to having non-PODs types declared as `struct` rather than `class`? – supercat Feb 18 '13 at 16:15

From the C++ FQA Lite:

> [16.4] Why should I use new instead of trustworthy old malloc()?

> FAQ: new/delete call the constructor/destructor; new is type safe, malloc is not; new can be overridden by a class.
>
> FQA: The virtues of new mentioned by the FAQ are not virtues, because constructors, destructors, and operator overloading are garbage (see what happens when you have no garbage collection?), and the type safety issue is really tiny here (normally you have to cast the void* returned by malloc to the right pointer type to assign it to a typed pointer variable, which may be annoying, but far from "unsafe").
>
> Oh, and using trustworthy old malloc makes it possible to use the equally trustworthy & old realloc. Too bad we don't have a shiny new operator renew or something.
>
> Still, new is not bad enough to justify a deviation from the common style used throughout a language, even when the language is C++. In particular, classes with non-trivial constructors will misbehave in fatal ways if you simply malloc the objects. So why not use new throughout the code? People rarely overload operator new, so it probably won't get in your way too much. And if they do overload new, you can always ask them to stop.

Sorry, I just couldn't resist. :)

answered Oct 8 '08 at 20:24

Matthias Benkard
13.1k 1 30 43

5 That is a *riot*! Thanks. – dmckee Feb 10 '09 at 14:02

3 I cannot take this comment serious as it clearly projects the author's biased against C++. C++ is a language used to create performance oriented software, and a garbage collector could only be detrimental to its objective. I disagree with your entire answer! – Miguel Nov 26 '15 at 21:23

@Miguel You missed the joke. – Dan Apr 11 '16 at 17:29

Always use new in C++. If you need a block of untyped memory, you can use operator new directly:

```
void *p = operator new(size);
   ...
operator delete(p);
```

edited Oct 9 '08 at 10:05

answered Oct 8 '08 at 19:54

Ferruccio
70.2k 31 176 268

2 interesting, i always just allocated an array of unsigned char when i need a raw data buffer like this. – Greg Rogers Oct 8 '08 at 19:57

Careful the semmantics should be like this: p_var = new type(initializer); Not size. – Brian R. Bondy Oct 8 '08 at 20:04

11 Not if you call operator new directly, then it takes the number of bytes to allocate as a parameter. – Ferruccio Oct 8 '08 at 20:05

1 Hrm not sure, I've never heard of this syntax. – Brian R. Bondy Oct 8 '08 at 20:06

8 The opposite of `operator new` is `operator delete`. It is not a well defined action to call `delete` on an expression with type `void*`. – Charles Bailey Oct 8 '08 at 20:16

Use **`malloc`** and **`free`** *only* for allocating memory that is going to be managed by c-centric libraries and APIs. Use **`new`** and **`delete`** (and the **`[]`** variants) for everything that you control.

edited Oct 8 '08 at 20:01

answered Oct 8 '08 at 19:51

dmckee
68.9k 17 105 189

8 Also notice that well written C library will hide malloc and free internally, this is how the C programmer should work. – Dacav Aug 13 '10 at 18:26

@dmckee do you have an example of C++ using c-centric libraries by malloc and free? – milesma Jul 31 '13 at 0:44

@Dacav: If a C function will accept a pointer to an object that it will need to keep using after the function returns, and the caller will have no way of knowing when the object is still needed, it would be perfectly reasonable for the function to specify that the pointer must have been created with `malloc`. Likewise if a function like `strdup` needs to create an object and return it to a caller, it is perfectly reasonable to specify that the caller must call `free` on the object when it is no longer needed. How could such functions avoid exposing their use of malloc/free to the caller? – supercat Apr 6 '15 at 17:09

@supercat, there's something inherently wrong in having a C function accept a pointer to objects, since C is not aware of objects at all. In general I believe the best approach is having semantic wrappers around allocation/deallocation also in C. It can be still acceptable, but less flexible, if a C library is asking the caller to pre-allocate and/or deallocate memory. If a C function is doing this and claiming ownership on the allocated memory, you are implicitly required to allocate it with malloc. – Dacav Apr 6 '15 at 20:30

new vs malloc()

1) `new` is an **operator**, while `malloc()` is a **function**.

2) `new` calls **constructors**, while `malloc()` does not.

3) `new` returns **exact data type**, while `malloc()` returns **void ***.

4) `new` never returns a **NULL** (will throw on failure) while `malloc()` returns NULL

5) Reallocation of memory not handled by `new` while `malloc()` can

answered Nov 26 '15 at 10:06

Yogeesh H T
838 7 10

1 Hi , For point 4) , new can be instructed to return NULL on failure. `char* ptr = new (std::nothrow) char [323232];` – Singh Oct 7 '16 at 8:54

6) new creates from constructor arguments, while malloc uses size. – Evan Moran Oct 16 '16 at 15:03

There is one big difference between `malloc` and `new`. `malloc` allocates memory. This is fine for C, because in C, a lump of memory is an object.

In C++, if you're not dealing with POD types (which are similar to C types) you must call a constructor on a memory location to actually have an object there. Non-POD types are very common in C++, as many C++ features make an object automatically non-POD.

`new` allocates memory *and* creates an object on that memory location. For non-POD types this means calling a constructor.

If you do something like this:

```
non_pod_type* p = (non_pod_type*) malloc(sizeof *p);
```

The pointer you obtain cannot be dereferenced because it does not point to an object. You'd need to call a constructor on it before you can use it (and this is done using placement `new`).

If, on the other hand, you do:

```
non_pod_type* p = new non_pod_type();
```

You get a pointer that is always valid, because `new` created an object.

Even for POD types, there's a significant difference between the two:

```
pod_type* p = (pod_type*) malloc(sizeof *p);
std::cout << p->foo;
```

This piece of code would print an unspecified value, because the POD objects created by `malloc` are not initialised.

With `new`, you could specify a constructor to call, and thus get a well defined value.

```
pod_type* p = new pod_type();
std::cout << p->foo; // prints 0
```

If you really want it, you can use use `new` to obtain uninitialised POD objects. See this other answer for more information on that.

Another difference is the behaviour upon failure. When it fails to allocate memory, `malloc` returns a null pointer, while `new` throws an exception.

The former requires you to test every pointer returned before using it, while the later will always produce valid pointers.

For these reasons, in C++ code you should use `new`, and not `malloc`. But even then, you should not use `new` "in the open", because it acquires resources you need to release later on. When you use `new` you should pass its result immediately into a resource managing class:

```
std::unique_ptr<T> p = std::unique_ptr<T>(new T()); // this won't leak
```

edited Nov 1 '11 at 18:33

community wiki
2 revs
R. Martinho Fernandes

There are a few things which `new` does that `malloc` doesn't:

1. `new` constructs the object by calling the constructor of that object
2. `new` doesn't require typecasting of allocated memory.

3. It doesn't require an amount of memory to be allocated, rather it requires a number of objects to be constructed.

So, if you use `malloc` , then you need to do above things explicitly, which is not always practical. Additionally, `new` can be overloaded but `malloc` can't be.

answered Jan 15 '14 at 14:45
herohuyongtao
28k 11 73 100

---

if you are using c++ then try to use new/delete instead of malloc/calloc as they are operator its self compared to malloc/calloc for them you used to include another header for that.so don't mix two different languages in single coding.their work is similar in every manner both allocates the memory dynamically from heap segment in hash table.

answered Apr 2 '14 at 9:13
user3488100
31 1

If you have C code you want to port over to C++, you might leave any malloc() calls in it. For any new C++ code, I'd recommend using new instead.

answered Oct 8 '08 at 19:52
Fred Larson
39.7k 8 81 112

If you work with data that doesn't need construction/destruction and requires reallocations (e.g., a large array of ints), then I believe malloc/free is a good choice as it gives you realloc, which is way faster than new-memcpy-delete (it is on my Linux box, but I guess this may be platform dependent). If you work with C++ objects that are not POD and require construction/destruction, then you must use the new and delete operators.

Anyway, I don't see why you shouldn't use both (provided that you free your malloced memory and delete objects allocated with new) if can take advantage of the speed boost (sometimes a significant one, if you're reallocing large arrays of POD) that realloc can give you.

Unless you need it though, you should stick to new/delete in C++.

edited Apr 9 '13 at 12:35

answered Apr 9 '13 at 12:23
PSkocik
16.9k 3 27 52

From a lower perspective, new will initialize all the memory before giving the memory whereas malloc will keep the original content of the memory.

answered Aug 14 '11 at 20:31
Peiti Peter Li
845 3 15 30

3 new does not in general initialize memory, although there are ways to make that happen: see stackoverflow.com/questions/2204176/… for one discussion about it. – wjl Sep 30 '11 at 14:54

The new and delete operators can operate on classes and structures, wheres malloc and free only work with blocks of memory that need to be cast.

Using new/delete will help to improve your code as you will not need to cast allocated memory to the required data structure.

answered Oct 8 '08 at 20:42
selwyn
781 2 7 19

`new` will initialise the default values of the struct and correctly links the references in it to itself.

E.g.

```
struct test_s {
    int some_strange_name = 1;
    int &easy = some_strange_name;
}
```

So `new struct test_s` will return an initialised structure with a working reference, while the malloc'ed version has no default values and the intern references aren't initialised.

answered Dec 14 '16 at 15:46

lama12345
1,087 1 10 12

In the following scenario, we can't use new since it calls constructor.

```
class  B  {
private:
    B *ptr;
    int x;
public:
    B(int n)  {
        cout<<"B: ctr"<<endl;
        //ptr = new B;  //keep calling ctr, result is segmentation fault
        ptr = (B *)malloc(sizeof(B));
        x = n;
        ptr->x = n + 10;
    }
    ~B()  {
        //delete ptr;
        free(ptr);
        cout<<"B: dtr"<<endl;
    }
};
```

edited Aug 17 '12 at 7:26

HackedByChinese
33.2k 5 98 128

answered Aug 17 '12 at 0:54

Barry
19 2

---

malloc() is used to dynamically assign memory in C while the same work is done by new() in c++. So you cannot mix coding conventions of 2 languages. It would be good if you asked for difference between calloc and malloc()

answered Jul 26 '12 at 5:41

Hitesh Ahuja
13 5

1 You *can* (but almost always shouldn't) use `malloc` in C++. – interjay Jul 26 '12 at 13:06

1 You also missed the prime point that you should aim to avoid dynamic memory allocation, unless doing so through smart pointers. You are just setting your self up for pain other wise – thecoshman Oct 11 '12 at 10:55
