

[xavier roche's homework](#)

random thoughts and sometimes pieces of code

- [RSS](#)

Search
Navigate... ▼

- [Blog](#)
- [Archives](#)

A Basic Glance at the Virtual Table

May 9th, 2014

Virtual Functions, vtable And Other Monsters

C++ inheritance, [virtual functions](#), and [virtual tables](#) are sometimes seen as obscure beasts, something most people will not mess with.

Casting an object to another one or calling a virtual function are probably two of the most common operations you can think of in C++, so it is a shame that such *simple* operations are not fully understood by everyone.

I have been programming in C for many years, and one of the benefits of C is that there is no real magic: when you define a structure, you can *see* the layout in memory, you can *feel* the pointers being padded, and every other operations, such as casting a pointer to another one (something you should only do with great care, by the way) are just re-interpreting memory addresses from one type to another one or shifting addresses a bit. No magic. No surprises. Accessing the second member of a pointer array ? This is just basic pointer arithmetic (indirect move with shifting and index, something that even the venerable [68000](#) processor had if my memory is correct)

Some people go straight in [C++](#) (or even in [Java](#)) when they start learning programming – which is in my point of view a bad thing – and tend to lose their focus on what's really happening behind the curtains (ie. inside the memory)

But things are generally much simpler than you thought.

Back To Basis

Let's first have a look at a very simple object. A C++ one:

```
1 class A {
2 private:
3   int id;
4
5 public:
6   A(): id(42) {
7     printf("yay, my address is %p, and my id is %d!\n", this, this->id);
8   }
9 };
```

This class contains only a constructor, and has one member (an integer). These two things are not of the same nature: variable members and functions are of different kind (yes, this is a really smart and insightful remark, I know).

To make things clearer, we can view this class as a C-structure, and its functions as global functions taking a (hidden) “this” member as first argument (the pointer to the structure itself). The instance members can be accessed implicitly, without the `this->` pointer indirection, by the way: C++ will add the missing bits.

The code above can be rewritten in **pure C** as follows:

```
1 // this is the actual "class A" instance data layout!
2 struct A {
3   int id;
4 };
5
6 // this is the class A equivalent constructor (a global function)
7 void A_A(struct A* this) {
8   this->id = 42;
9   printf("yay, my address is %p, and my id is %d!\n", this, this->id);
10 }
```

... and, rather than just declaring:

```
1 A a;
```

... you'll need to call the *constructor* manually:

```
1 struct A a;
2 A_A(&a); // call constructor manually!
```

Here it is! C++ in plain C! Of course a C++ compiler will *hide everything behind the curtains* when using regular classes, but the actual *compiled code* will be more or less the same.

If you add more members and/or more methods in this class, the principle is the same: extend the structure layout, and/or add more global functions that take an hidden *this* first argument.

Quizz: what is the size of a `class A` object instance ? The answer is pretty simple: the class only contains one member (`id`), which is an integer (32-bit here) ; the `sizeof(A)` is therefore 4 (bytes). No magic.

Quizz: what will be the size of a `class A` object instance if a regular member function is added ? The answer is pretty simple, too: its size won't change, as it still contains only one member – only an additional global function will be added in the code.

What About Inheritance ? This Seems a Bit More Complicated!

No, no, *not at all*.

Let's take a simple (yet representative) example: our previous `A` class, another `B` class with a single "age" member, and at last a final class `C` with a member "mode", but inheriting from `A` and `B`.

```
1 class A {
2 public:
3     int id;
4
5     A(): id(42) {
6         printf("A: yay, my address is %p, my size is %zu, and my id is %d!\n",
7             this, sizeof(*this), this->id);
8     }
9 };
10
11 class B {
12 public:
13     int age;
14
15     B(): age(7) {
16         printf("B: yay, my address is %p, my size is %zu, and my age is %d!\n",
17             this, sizeof(*this), this->age);
18     }
19 };
20
21 class C: public A, public B {
22 public:
23     int mode;
24
25     C(): mode(-1) {
26         printf("C: yay, my address is %p, my size is %zu, my id, age and mode are %d, %d, %d!\n",
27             this, sizeof(*this), this->id, this->age, this->mode);
28     }
29 };
```

Instantiating an object from `C` (`C c;`) will print something like:

```
1 A: yay, my address is 0x73620f9f5c20, my size is 4, and my id is 42!
2 B: yay, my address is 0x73620f9f5c24, my size is 4, and my age is 7!
3 C: yay, my address is 0x73620f9f5c20, my size is 12, my id, age and mode are 42, 7, -1!
```

The `class C` constructor called first the upper inherited classes constructors (`class A` constructor and `class B` constructor), and therefore prints their information first. Here again, C++ hides everything behind curtains: the calls to base classes constructors are generated automatically, before your actual `class C` constructor code.

You can note that the address seen by the constructors of `A` and `C` are identical, *but not the one seen by B*. How can it be ?

Let's rewrite everything in plain C:

```

1 struct A {
2     int id;
3 };
4
5 // class A constructor
6 void A_A(struct A* this) {
7     this->id = 42;
8     printf("A: yay, my address is %p, my size is %zu, and my id is %d!\n",
9           this, sizeof(*this), this->id);
10 }
11
12 struct B {
13     int age;
14 };
15
16 // class B constructor
17 void B_B(struct B *this) {
18     this->age = 7;
19     printf("B: yay, my address is %p, my size is %zu, and my age is %d!\n",
20           this, sizeof(*this), this->age);
21 }
22
23 struct C {
24     struct A a;
25     struct B b;
26     int mode;
27 };
28
29 void C_C(struct C *this) {
30     // manually call base classes constructors first
31     A_A(&this->a);
32     B_B(&this->b);
33     // then, user code
34     this->mode = -1;
35     printf("C: yay, my address is %p, my size is %zu, my id, age and mode are %d, %d, %d!\n",
36           this, sizeof(*this), this->a.id, this->b.age, this->mode);
37 }

```

The struct C layout is pretty straightforward: the first element is the struct A members – the reason why constructors from class A and class C did see the same address, the second member is the member(s) of class B, and finally comes the member(s) of C.

In both C++ and C variants,

- the size of C is 12 bytes (the three integers)
- the location of A within C is at the beginning (the first four bytes)
- the location of B within C is after the area dedicated to A (the next four bytes)
- the location of members specific to C then follows

Casting will provide the same behavior, by the way:

- `static_cast<A*>(&c) == &(c)->a == 0x73620f9f5c20`
- `static_cast<B*>(&c) == &(c)->b == 0x73620f9f5c24`
- `static_cast<C*>(&c) == &c == 0x73620f9f5c20`

Okay, But Virtual Functions ? Can You Do That In Plain C ?

Well, let's consider a simple example: simply add a virtual function print to our previous sample, aiming to print the object class name and parameters. The idea is that even if we cast the high-level class C instance into A* or B*, we'll see the same behavior.

```

1 class A {
2 public:
3     int id;
4
5     A(): id(42) {
6         printf("A: yay, my address is %p, my size is %zu, and my id is %d!\n",
7               this, sizeof(*this), this->id);
8     }
9
10    virtual void print() {
11        printf("I am A(%d)\n", id);
12    }
13 };
14
15 class B {
16 public:
17     int age;

```

```

18
19 B(): age(7) {
20     printf("B: yay, my address is %p, my size is %zu, and my age is %d!\n",
21           this, sizeof(*this), this->age);
22 }
23
24 virtual void print() {
25     printf("I am B(%d)\n", age);
26 }
27 };
28
29 class C: public A, public B {
30 public:
31     int mode;
32
33 C(): mode(-1) {
34     printf("C: yay, my address is %p, my size is %zu, my id, age and mode are %d, %d, %d!\n",
35           this, sizeof(*this), this->id, this->age, this->mode);
36 }
37
38 virtual void print() {
39     printf("I am C(%d, %d, %d)\n", id, age, mode);
40 }
41 };

```

If we create an instance of C called c (C c;), the following calls will produce exactly the same output:

- c.print();
- static_cast<A*>(&c)->print();
- static_cast<B*>(&c)->print();
- static_cast<C*>(&c)->print();

Where is the magic ? Let's first have a look at the output, when creating this object:

```

1 A: yay, my address is 0x726fa25b7c00, my size is 16, and my id is 42!
2 B: yay, my address is 0x726fa25b7c10, my size is 16, and my age is 7!
3 C: yay, my address is 0x726fa25b7c00, my size is 32, my id, age and mode are 42, 7, -1!

```

We can see that A and B are not anymore 4 bytes, but 16. And C is 32 bytes! Somehow, the c++ compiler did add some bytes in our objects without telling us!

Let's have a look at each object instance actual contents within each classes, by dumping data as pointers in constructors, to figure ou *what has been added* inside our beloved ~~structures~~ classes:

```

1 // simple dump function
2 static void print_object(const char *name, void *this_, size_t size) {
3     void **ugly = reinterpret_cast<void**>(this_);
4     size_t i;
5     printf("created %s at address %p of size %zu\n", name, this_, size);
6     for(i = 0 ; i < size / sizeof(void*) ; i++) {
7         printf("  pointer[%zu] == %p\n", i, ugly[i]);
8     }
9 }

```

And using print_object(__FUNCTION__, this, sizeof(*this)); inside each constructor.

Remember that the constructors are called as follows when creating an instance of C:

- base class A constructor is called magically (this points to the beginning of the allocated C structure)
- base class B constructor is called magically (this starts at 16 bytes from the beginning of the allocated C structure)
- top level C constructor user code is executed (this points to the beginning of the allocated C structure)

Here's the result:

```

1 created A at address 0x7082f962ccb0 of size 16
2   pointer[0] == 0x400f20
3   pointer[1] == 0x2a
4 created B at address 0x7082f962ccc0 of size 16
5   pointer[0] == 0x400f00
6   pointer[1] == 0x7
7 created C at address 0x7082f962ccb0 of size 32
8   pointer[0] == 0x400ed0
9   pointer[1] == 0x2a

```

```

10 pointer[2] == 0x400ee8
11 pointer[3] == 0xffffffff00000007

```

We can see that:

- A contains *an unknown additional pointer* (0x400f20), and the id variable (0x2a == 42) padded to 8 bytes
- B contains *another unknown additional pointer* (0x400f00), and the age variable (7) padded to 8 bytes
- C embeds A and B, and the mode variable (-1): these are seen as 0x2a, and 0xffffffff00000007 (00000007 for the age variable, and ffffffff for the mode variable: -1 is 0xffffffff in [two complement](#)'s world) – and the two additional pointers

So it *seems* that the structure size change reason is that an hidden pointer member (8 bytes in the 64-bit world) has been added to the class instance – and the compiler probably padded the structure to 16 bytes, because you want to have 8-byte alignment when dealing with arrays: if you have an array of two objects, the second object hidden pointer has to be padded to 8 bytes, enforcing the structure to be a multiple of 8 bytes. The 4 bytes used for padding are unused, but useful to maintain the alignment.

Hey, wait! The two pointers are different, when printed inside the C constructor! Yes, good catch: the additional A pointer which was 0x400f20 inside the `class A` constructor has been changed into 0x400ed0 inside the `class C` constructor, and the additional B pointer which was 0x400f00 inside the `class B` constructor has been changed into 0x400ee8 inside the `class C` constructor.

Oh, and a last note: C is *only* 32 bytes.

- there is no additional hidden pointer for C!
- the compiler apparently used the padding area of `class B` for the mode variable (smart, isn't it ?)

Okay, so **two magical pointers** have been added because we defined a virtual function. What does it mean ?

Let's go back to basic: *how would you implement this class hierarchy in pure C ?*

Here's what you may do:

```

1 struct A {
2     int id;
3 };
4
5 // class A constructor
6 void A_A(struct A *this) {
7     this->id = 42;
8     print_object(__FUNCTION__, this, sizeof(*this));
9 }
10
11 void A_print(struct A *this) {
12     printf("I am A(%d)\n", this->id);
13 }
14
15 struct B {
16     int age;
17 };
18
19 // class B constructor
20 void B_B(struct B *this) {
21     this->age = 7;
22     print_object(__FUNCTION__, this, sizeof(*this));
23 }
24
25 void B_print(struct B *this) {
26     printf("I am B(%d)\n", this->age);
27 }
28
29 struct C {
30     struct A a;
31     struct B b;
32     int mode;
33 };
34
35 void C_C(struct C *this) {
36     // manually call base classes constructors first
37     A_A(&this->a);
38     B_B(&this->b);
39     // then, user code
40     this->mode = -1;
41     print_object(__FUNCTION__, this, sizeof(*this));
42 }
43
44 void C_print(struct C *this) {
45     printf("I am C(%d, %d, %d)\n", this->a.id, this->b.age, this->mode);
46 }

```

Okay, looks more or less what we had previously. But what about the *virtual* function ? We have to be sure that we call the right function, even if we only have a A* or a B* pointer (and we don't *know* that this is actually a C object class behind).

A solution can be to add a **function pointer** for every *virtual function* in each classes, and initialize inside the constructor!

Something like this:

```

1 struct A {
2     void (*print)(struct A *this);
3     int id;
4 };
5
6 // class A constructor
7 void A_A(struct A *this) {
8     this->print = A_print;
9     this->id = 42;
10    print_object(__FUNCTION__, this, sizeof(*this));
11 }
12
13 struct B {
14     void (*print)(struct B *this);
15     int age;
16 };
17
18 // class B constructor
19 void B_B(struct B *this) {
20     this->print = B_print;
21     this->age = 7;
22    print_object(__FUNCTION__, this, sizeof(*this));
23 }
24
25 void B_print(struct B *this) {
26    printf("I am B(%d)\n", this->age);
27 }
28
29 struct C {
30     struct A a;
31     struct B b;
32     void (*print)(struct C *this);
33     int mode;
34 };
35
36 void C_C(struct C *this) {
37     // manually call base classes constructors first
38     A_A(&this->a);
39     B_B(&this->b);
40     // then, user code
41     this->print = C_print;
42     this->a.print = C_print; // we patch our own A->print function !!
43     this->b.print = C_print; // and we need to do it for B->print too!!
44     this->mode = -1;
45     print_object(__FUNCTION__, this, sizeof(*this));
46 }
47
48 void C_print(struct C *this) {
49    printf("I am C(%d, %d, %d)\n", this->a.id, this->b.age, this->mode);
50 }

```

What did we do here ?

- A and B constructors initialize their versions of print
- C constructor initialize its own version of print after A and B constructors, and **overrides the one inside A and B**

Neat, isn't it ?

Now, whatever pointer you have, these three lines will produce identical behavior:

- a->print(a);
- b->print(b);
- c->print(c);

Hey, by the way, *we do not need an additional pointer for C*, because we already have the correct version in two locations: a.print and b.print. Let's remove this useless pointer in C (and the useless initialization), and use:

- a->print(a);
- b->print(b);
- c->a.print(c);

There is a little problem, however, when using a C instance cast in A* or B*. When you call `a->print(a)`, the `a` pointer is of type A*. And therefore the assignment `this->a.print = C_print` will produce a warning. Similarly, when you call `b->print(b)`, the `this` pointer will have a type of B*, and not only it is of the wrong type, but it has a different address, as `c->b` is located *sixteen bytes* after the beginning of C.

We need *specialized* versions with correct casts!

- casting from A* to C* is straightforward: *we know* that the address is the same, and the type can be safely cast
- casting from B* to C* requires to adjust the address to find the beginning of C, and a cast (B is in the *middle* of C, remember)

Here we go:

```

1 void C_print(struct C *this) {
2     printf("I am C(%d, %d, %d)\n", this->a.id, this->b.age, this->mode);
3 }
4
5 void CA_print(struct A *this) {
6     C_print((struct C*) this);
7 }
8
9 void CB_print(struct B *this) {
10    // we know that this points after the struct A layout
11    // so let's fix the pointer by ourselves!
12    size_t offset = (size_t) &((struct C*) NULL)->b;
13    C_print((struct C*) ( (char*) this ) - offset );
14 }
15
16 void C_C(struct C *this) {
17    // manually call base classes constructors first
18    A_A(&this->a);
19    B_B(&this->b);
20    // then, override functions
21    this->a.print = CA_print; // we patch our own function !!
22    this->b.print = CB_print; // and we need to do it for B too!!
23    // this->print = C_print; // useless
24    // then, user code
25    this->mode = -1;
26    print_object(__FUNCTION__, this, sizeof(*this));
27 }

```

With these adjustments, things are smooth:

- `c.a.print(&c.a) ==> I am C(42, 7, -1)`
- `c.b.print(&c.b) ==> I am C(42, 7, -1)`
- `c.print(&c) ==> I am C(42, 7, -1)`

Perfect, isn't it ?

Not yet: what if we add more *virtual functions* in our structures ? This will increase the object size, and require more pointer initialization in constructors! What a waste of space and time: we are assigning *the same pointers* each time. Could we *prepare* the set of pointers for each classes in a static, global structure, and just initialize a *pointer to these static data* inside the constructors ? Neat, isn't it ?

Here's the modified version – let's call the global structure tables containing specialized function pointers `vtb1`, as in *virtual table*, because this is actually what they are: table for virtual functions!

```

1 // type for global virtual function table structure A
2 // this structure enumerates all virtual functions of A
3 struct A;
4 struct table_A {
5     void (*print)(struct A *this);
6 };
7
8 struct A {
9     const struct table_A *vtb1;
10    int id;
11 };
12
13 void A_print(struct A *this) {
14     printf("I am A(%d)\n", this->id);
15 }
16
17 // the global, static table data for A
18 // this structure enumerates all virtual functions of B
19 static const struct table_A table_A_for_A = { A_print };
20
21 void A_A(struct A *this) {
22     this->vtb1 = &table_A_for_A;
23     this->id = 42;
24     print_object(__FUNCTION__, this, sizeof(*this));

```

```

25 }
26
27 // type for global virtual function table structure B
28 struct B;
29 struct table_B {
30     void (*print)(struct B *this);
31 };
32
33 struct B {
34     const struct table_B *vtbl;
35     int age;
36 };
37
38 void B_print(struct B *this) {
39     printf("I am B(%d)\n", this->age);
40 }
41
42 // the global, static table data for B
43 static const struct table_B table_B_for_B = { B_print };
44
45 void B_B(struct B *this) {
46     this->vtbl = &table_B_for_B;
47     this->age = 7;
48     print_object(__FUNCTION__, this, sizeof(*this));
49 }
50
51 struct C {
52     struct A a;
53     struct B b;
54     int mode;
55 };
56
57 void C_print(struct C *this) {
58     printf("I am C(%d, %d, %d)\n", this->a.id, this->b.age, this->mode);
59 }
60
61 void CA_print(struct A *this) {
62     C_print((struct C*) this);
63 }
64
65 void CB_print(struct B *this) {
66     // we know that this points after the struct A layout
67     // so let's fix the pointer by ourselves!
68     size_t offset = (size_t) &((struct C*) NULL)->b;
69     C_print((struct C*) ( (char*) this ) - offset );
70 }
71
72 // the global, static tables data for A and B structures, for C
73 static const struct table_A table_A_for_C = { CA_print };
74 static const struct table_B table_B_for_C = { CB_print };
75
76 void C_C(struct C *this) {
77     // manually call base classes constructors first
78     A_A(&this->a);
79     B_B(&this->b);
80     // Override virtual function pointers that have just been initialized by
81     // A and B constructors with our own version. This allows to override all
82     // possible virtual functions in base classes!
83     this->a.vtbl = &table_A_for_C;
84     this->b.vtbl = &table_B_for_C;
85     this->mode = -1;
86     print_object(__FUNCTION__, this, sizeof(*this));
87 }

```

Neat, isn't it ?

Well, I'm telling you a secret: *this is more or less what the C++ compiler does* (yes, behind the curtains) when implementing virtual functions. The virtual function tables are called *virtual tables*, and are static structures inside the compiled code. And inside each corresponding class constructor, additional code is *inserted* just between base class initialization calls, and user code, to set all virtual table pointers corresponding to the class type.

Some additional remarks on this design:

- if you call a virtual function *inside* a base class constructor, you will **not** call the top-level function version, because the virtual table pointer has not *yet* been initialized (it will be initialized *after* the end of the base class constructor)
- you need *one* static structure containing all necessary stuff for a given class
- you can put additional information on these virtual table structures, which can be useful!
 - the class name
 - type information functions (what is this object's real class ?), for example by implementing a hidden `getClassType()` virtual function

- dynamic cast support

... and this is exactly what most compiler do, actually: [Runtime Type Information](#) data is generally included as soon as a virtual table needs to be created for a given class (ie. you can use `dynamic_cast<>()`)

Now you know what's going on when you define a virtual function: rather than calling the function directly, an indirection through a virtual table is done.

This has some drawbacks:

- a little function call impact on performances, because you need to read the actual function pointer in a static table
- a potentially *huge* impact on performances because *you can not inline anymore function calls*

A Deeper Look

Let's dig a little bit more, using [GCC](#) version of the virtual tables.

If you run your program under [GDB](#) and examine an instance of class `C`, you'll see:

```
1 (gdb) p c
2 $1 = {<A> = {_vptr.A = 0x400e70 <vtable for C+16>, id = 42}, <B> = {_vptr.B = 0x400e88 <vtable for C+40>, age = 7}, mode = -1}
```

Note the `_vptr.A = 0x400e70` and `_vptr.B = 0x400e88` : these are the virtual tables.

What is behind these pointers ? Recent GCC releases allows you to dig a bit into an object's virtual table:

```
1 (gdb) info vtbl c
2 vtable for 'C' @ 0x400e70 (subobject @ 0x775bfbf98600):
3 [0]: 0x400b00 <C::print()>
4
5 vtable for 'B' @ 0x400e88 (subobject @ 0x775bfbf98610):
6 [0]: 0x400b34 <non-virtual thunk to C::print()>
```

Otherwise, you can easily guess that each vtables starts with the same kind of function array we created for our C version:

```
1 (gdb) p ((void**)0x400e70)[0]
2 $5 = (void *) 0x400b00 <C::print()>
3
4 (gdb) p ((void**)0x400e88)[0]
5 $9 = (void *) 0x400b34 <non-virtual thunk to C::print()>
```

Easter Egg

Now you understand a bit more the concept of vtable, let's see a dirty-but-fun hack (valid for GCC, possibly for Visual C++ too): how to *override* a specific object's virtual function table, and rewire virtual functions at runtime

```
1 // Our patched version of C::printf (or A::printf)
2 static void our_function(C *thiz) {
3     printf("I am now a new C(%d, %d, %d)!\n", thiz->id, thiz->age, thiz->mode);
4 }
5
6 // Our patched version of B::printf
7 static void our_function_from_B(B *thiz) {
8     // This is the same boring arithmetic than before, except that
9     // we use a fake C instance located at a random (but non-NULL) address
10    // to compute the offset. Note that the actual instance address will
11    // never be dereferenced, so do not fear illegal accesses!.
12    C *dummy = reinterpret_cast<C*>(0x42);
13    B *dummy_B = static_cast<B*>(dummy);
14    char *dummy_ptr = reinterpret_cast<char*>(dummy);
15    char *dummy_B_ptr = reinterpret_cast<char*>(dummy_B);
16    size_t offset = dummy_B_ptr - dummy_ptr;
17    char *ptr = reinterpret_cast<char*>(thiz);
18    C *thiz_C = reinterpret_cast<C*>(ptr - offset);
19    our_function(thiz_C);
20 }
21
22 int main(int argc, char **argv) {
23     // create an instance of C
24     C c;
25
26     // Hack the c instance and override its print() function!
```

```

27 void *our_vtbl[1] = { reinterpret_cast<void*>(our_function) };
28 void *our_vtbl2[1] = { reinterpret_cast<void*>(our_function_from_B) };
29 // patch for A* and C*
30 *reinterpret_cast<void**>(static_cast<A*>(&c)) = our_vtbl;
31 // patch for B*
32 *reinterpret_cast<void**>(static_cast<B*>(&c)) = our_vtbl2;
33
34 // get the pointers (note: GCC will optimize direct calls to c.printf)
35 A *pa = &c;
36 B *pb = &c;
37 C *pc = &c;
38
39 // and call printf() ... what will be printed ?
40 pa->print();
41 pb->print();
42 pc->print();
43
44 return 0;
45 }

```

And yes, your object was hacked:

```

1 I am now a new C(42, 7, -1)!
2 I am now a new C(42, 7, -1)!
3 I am now a new C(42, 7, -1)!

```

Neat, isn't it ? (yes, yes, not really portable though)

References

- You may have a look at the [Itanium C++ ABI, Virtual Table Layout](#) description (GCC [is using](#) the Itanium ABI for vtables, or at least something very close)
- For Windows coders, the [Reversing Microsoft Visual C++ Part II](#) page will be a goldmine, too

TL;DR: remember that C++ used to be *enhanced C* a long time ago, and most basic features can be easily implemented **and understood** through plain C code!

Posted by Xavier Roche May 9th, 2014 [c](#), [c++](#), [programming](#)

[Tweet](#)

[« A Story Of realloc \(And Laziness\) C Corner Cases \(and Funny Things\) »](#)

Comments




Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS **Paul Masurel** • 6 years ago

Neat ! Suggestion for your next blog post. How to use the curiously recurring pattern to avoid virtual methods perf drops, when being dynamic does not matter.

32  |  • [Reply](#) • [Share](#) ›**egovind** • 4 years ago

Nice explanation!

 |  • [Reply](#) • [Share](#) ›**Julie Wang** • 4 years ago

Nice Post! From which I learn a lot useful info! "there is no additional hidden pointer for C! The compiler apparently used the padding area of class B for the mode variable (smart, isn't it ?)" -- Yes, it's right that there is not additional hidden pointer for C but this might not because of compiler's management. If there is no data member in A,B,C, the size of C's object still equals to A+B. According to 'Inside the C++ object model', there would be N-1 additional virtual tables and virtual pointers if a class inherits from N other classes. Meanwhile, the inherited class is sharing the virtual table and virtual pointer with the left most base class. Thus the total hidden pointer number is N. ;)

 |  • [Reply](#) • [Share](#) ›**Björn De Meyer** • 5 years ago

Nice post, but you really should use the standard offsetof() macro in stead of
size_t offset = (size_t) &((struct C*) NULL)->b;

 |  • [Reply](#) • [Share](#) ›

Recent Posts

- [Redirecting Stdio Is Hard](#)
- [I Found a Bug in Strncat\(\)](#)
- [Coucal, Cuckoo-hashing-based Hashtable With Stash Area C Library.](#)
- [C Corner Cases \(and Funny Things\)](#)
- [A Basic Glance at the Virtual Table](#)
- [A Story of Realloc \(and Laziness\)](#)
- [Security Is Painful](#)
- [What Books Really Helped You ?](#)
- [Embrace Unicode \(and Do Not Worry\)](#)
- [What Are Your GCC Flags ?](#)
- [Fancy Standalone Visual C++ Compiler](#)
- [I Do Not Want Your Search Bar](#)
- [Everything You Always Wanted to Know About Fsync\(\)](#)
- [Coding Is Like Cooking](#)
- [Creating Deletable and Movable Files on Windows](#)
- [An Unusual Case of Case \(/switch\)](#)
- [Replacing JNI Crashes by Exceptions on Android](#)
- [HTTrack on Android](#)
- [MD5 Is Your Friend](#)
- [So, I Made a Hashtable](#)

GitHub Repos

- [htrack](#)
HTTrack Website Copier, copy websites to your computer (Official repository)
- [squandersnitch](#)
Expensive allocation snitch LD_PRELOAD module
- [coffeecatch](#)
CoffeeCatch, a tiny native POSIX signal catcher (especially useful for JNI code on Android/Dalvik)
- [coucal](#)
Cuckoo Hashtable C Library
- [enhancedminmax](#)
Enhanced version of std::min and std::max, supporting references and different integral types signedness
- [checkedcast](#)
Runtime safe type cast check templated decorator
- [stringswitch](#)
Switch/Case strings using constexpr long hash
- [htrack-android](#)
HTTrack Website Copier, copy websites to your computer - Android GUI (Official repository)
- [htrack-windows](#)
HTTrack Website Copier, copy websites to your computer - Windows GUI (Official repository)
- [fastlzlib](#)
ZLib-like wrapper around LZ4/Fastlz compression library.

[@xroche](#) on GitHub

Copyright © 2017 - Xavier Roche - Powered by [Octopress](#)