

Re: run-time vs compile-time

Source: <http://coding.derkeiler.com/Archive/C/ CPP/alt.comp.lang.learn.c-cpp/2004-09/0508.html>

From: Jonathan Mcdougall (jonathanmcdougall_at_DELYahoo.ca)

Date: 09/10/04

Date: Fri, 10 Sep 2004 07:38:17 -0400

> *My confusion is from here: C++ says that static allocation, such as*
>
> *static int i;*
>
> *is binding at compile-time, while dynamic allocation, such as*
>
> *int* pi = new int;*
>
> *is binding at run-time.*

The C++ standard is completely silent on the way the compiler and system implement allocation or "binding". C++ mandates behavior, not implementation. An implementation would be free to implement the stack and the heap the same way, as long as everything behaves as the standard stated. That is important.

However...

>*I understand now that for i, compiler can put an*
> *offset related to some location (like stack base) somewhere. But I still*
> *have some confusion about dynamic binding. To me, the compiler may put an*
> *offset from heap to pi.*

It is possible to use "offsets" when using the stack, since the system uses a stack pointer to know exactly what starting point you are using.

The offsets are always the same, but the starting point will change. The thing is, you only have one starting point in the heap, which is the start of the heap. Consider it to be address 1.

When you allocate an int on the heap, it is allocated at address 1. The next free chunk will be at address 5 (if an int is four bytes). If you allocate 10 ints on the heap, the the next free chunk will be at address 41. If you then free int number 2 and 5, you get some empty spaces scattered in the heap. If you then allocate another int, it will have address 5, which is the address of the first int you freed.

The heap is a `_disorganized_` system. You never know what is free and

what is allocated, that is why the compiler has no way, nor the system, to know in advance the address of a heap allocated object. It depends on what is in the heap right now.

If you run two instances of your program at the same time, both will have the same "offsets" on stack, that is, their structure on the stack will be exactly the same (if they get the same input of course). But they will obviously get different addresses from the heap.

Let's try something else.

You need to understand a little more about how a computer works. Take for example

```
void f(int a, int b)
{
}
```

f() resides somewhere in memory as machine code. When it is executed, that is, when the instruction pointer gets to its address, the values passed to f() are pushed on the stack and you can access 'a' by doing something like *(stack_pointer) and 'b' by doing *(stack_pointer + sizeof(int)). That is what I meant by 'offset'. The compiler/linker have no idea about where the stack_pointer will point at the time of executing f(), but it knows that if it goes to the address (stack_pointer + sizeof(int)), it will find 'b'.

The stack has a very rigid structure, since it must be constructed from the bottom to the top, by pushing and popping values. This structure allows the system to keep a pointer to the current position in the stack.

The heap is not that organized. In modern operating systems, an application has a given amount of memory it can use as it wishes. In that space, it can allocate and free memory independently of what is going on on the stack.

Since a stack is last-in-first-out, it is impossible to force some memory to be kept. Imagine a pile of plates. When you call a function, you push, for example, 5 plates on the top of the current stack. When that function returns, you remove these plates to get back to where you were. If you had some important informations you wanted to keep in one of these 5 plates, it is now lost.

The solution is to put that important information/plate aside, not in the stack of plates. The system then tries to find a free space on the side, in what is called the heap. If it finds the space, it returns a pointer to that memory space.

But since the heap is "disorganized", that is, your application could free and allocate memory in a random order, it is impossible for the compiler or even the system to know in advance what memory will be used.

To make the explanation simpler, imagine the heap as a linear chunk of memory, say of 10 mb. When an application request some memory on the heap, to bypass the rigid stack memory management, the system does a linear search in that chunk to find some free memory and returns a pointer to that place. It then marks that place as "allocated".

If 10 applications are running at the same time, requesting and freeing memory randomly (from the system's point of view), the memory will become fragmented. Allocated and free memory will be randomly scattered in the 10mb chunk.

At that point, when you decide to run your application, you have absolutely no way to know anything about the current memory state. You only ask the system for a free space and the system answers with a valid space, or fails.

That is why stack-allocated objects are taken care of by the system, since it knows exactly what is allocated. By having the stack of plates before you, you know how many plates you have to take off to get back to where you were and to free the memory used by objects. In the heap, however, it is impossible to know when the memory must be freed. And it is your task, as the programmer, to keep track of every chunk of memory you allocated in order to be able to free them.

*>But why we call it run-time binding? To me, they
> all decide at compile-time, that is, compiler record an offset from stack or
> heap.*

The heap is a disorganized pool of memory. Asking for free memory today will not return the same thing as tomorrow. It depends on what is in the heap right now. The stack, however, will always have the same structure and the same behavior. Arguments will always be sizeof(int) bytes after the current stack pointer, wherever the stack pointer points.

I hope it is a little clearer now. If not, continue asking and wait for somebody else to answer, since I don't know if I will be able to be clearer than that.

Jonathan