Memory: Stack & Heap

The new material in this first part of the course focuses on memory management, since this is one of the most important differences between Java and C++. We started with argument passing (by value, by reference, by const reference) and hidden arguments, because it is important to know when data are copied, and when new instances are created. We then expanded this to scoping, so that you would know when constructors and destructors are called. Now we want to focus on the question of where data are stored.

A quick review from CS270 (or its equivalent): Every process has a virtual memory space. Part of this is reserved for the operating system. The rest is divided into stack and heap. The stack starts at one end (typically the top), the heap at the other. The first function is always ‘main’, so the first frame is main’s frame. Main calls functions, which create new frames, which extend the stack.

What is in a frame? Some stuff the compiler uses to manage function calls (the program ctr, slots for register values, etc.). As a programmer you can ignore this and assume it works. But what you need to remember is that local variables are stored on the stack, in the frame of the function they are a part of.

A reminder of the scoping rule for local variables: local variables are allocated when the line that declares them is executed, and they go away when the function returns (or, more accurately, at the end of the block they are defined in, as signaled by }). More importantly, the constructor for a variable is invoked where it is declared. In theory, all objects have constructors; a default empty constructor is provided for any class without one.

There is one exception to this rule: return values. One might expect return values to be returned by value, so that the local copy would be destructed and a new copy constructed in the calling function. This would be semantically consistent with the principles of C++, but not very efficient. And efficiency is an important goal of C++. Therefore, an exception is made for return values: they are magically copied from the stack frame of the method/function to the returned value in the calling function, without triggering a destructor or constructor.

It is easy for Java programmers to forget that C++ has destructors, not just constructors. An instance’s destructor is called when it falls out of scope.

Consider the following example:

```cpp
using std::ifstream;
Complex Add(ifstream& istr)
{
    Complex c1, c2;
    C1.Read(istr);
    C2.Read(istr).
    C1.Add(c2);
    Return c1;
}
```
In the example above, how many destructors are called on the }? One. There are two local variables of type Complex: c1 and c2. ‘c2’ is destroyed when it falls out of scope. ‘c1’ is not, because it is the return value. The parameter is a reference, not a local variable, so no destructor is called for it. Had the argument been passed by value instead of by reference, it would have created a local copy that would have triggered a second destructor on the }.

Side note: that question was typical of a midterm question. For the exams, I will give you code and then ask you questions about it, such as how many destructors are called on a particular line of code, which could be the ending bracket (}).

I lot of code can be invoked by a bracket (}). I can’t tell you how often I have had functions crash when they return. You step through them in the debugger, they run fine, and then they crash on the return statement. Why? Because of a bug in one of the destructors…

Now, let’s go back and think about arguments to functions again. Where does the memory for a function argument live? It depends on the type of the argument.

If an argument is passed by value, it is a local variable and it lives in the stack frame of the called (not calling) function. It is created just before the function is called. This means that if you pass an object by value, it is copied and the object’s constructor is called before you enter the called function. It also means that the object’s destructor is called when the function returns.

If an argument is passed by reference or constant reference, then no new copy of the argument is made. So no constructor is called when the argument is passed, and no destructor is called at the end of the function, at least for the parameter. Where does the argument live? Depends…

OK, simple things can be important.
- Local variables are stored on the stack
  - They might be primitive data types
  - They might be complex objects
- Constructors are invoked
  - where local variables are declared
  - where an argument is passed by value
- Destructors are invoked
  - When local variables fall out of scope
  - When non-reference parameters (arguments) fall out of scope
- References are not local variables; no constructors or destructors are called

Semantically, this is different from Java. In Java, primitives are stored on the stack, and references to objects are stored on the stack. Objects are allocated on the heap. And of course, there are no destructors, only constructors.

Keep in mind that in a Java program, if you create an instance of complex, you are actually manipulating a reference to the object. In C++, if you declare an instance of complex, that is what you get: an instance of complex, not a reference to it. Put another way, it is possible for an object to be a local variable on the stack in C++; it is not in Java.

A bug that crops up in C++ but not Java is the invalid return value. For example, imagine I write the following:

```
Complex& Foo::Bar()
```
What happens? The return value is a reference to an instance of Complex. So we return c1 as a reference. But c1 lives in the stack frame of the Foo::Bar method. So it’s destructor is called when the function returns, and the stack frame it was living in is popped. So what is returned? A reference is an invalid (destroyed) variable. This is a bug.

The example above is legal C++ syntax, however. One of the reasons for asking the compiler for warnings is that many (but not all) compilers will warn you if you do this. The Gnu C++ compiler (the g++ command) does warn you about this if –Wall is used.

Note that to fix this bug, you only have to return an instance of Complex rather than a reference to one. (In other words, remove the & from the return type declaration). Return values are copied¹. Therefore returning a Complex value copies it, as in the code below.

```cpp
Complex Foo::Bar()
{
    Complex c1;
    // code to compute the value of c1;
    return c1;
}
```

So local variables live on stack. How do you put data on the heap?

Before we ask that, why would we want to put data on the heap? Two reasons:
1. Data on the stack falls out of scope and is reclaimed when the function that allocates is returns. This means the data cannot persist longer than the function. Data on the heap, on the other hand, is persistent.
2. The size of the stack frame is determined at compile time. Therefore, if data is to be put on the stack, the compiler must know its size at compile time. The size of data on the heap, on the other hand, can be determined at run time.

So we have reasons to want to put data on the heap. How do we do it? Via the “new” command. But before we talk about “new”, we need to introduce pointers.

Pointers are a data type. If X is a data type in C++ (any data type), then X* is a new data type called “pointer to X”. Thus I can write the following:

```cpp
Int* foo_ptr;
Penguin* p_ptr;
Penguin** indirect_ptr;
```

¹ In other words, return values are “returned by value”, but that is a confusing sentence.
So by the rule about any data type has a pointer data type, and pointers are data types, it is recursive. I can have a pointer to a pointer to a pointer… There are infinitely many data types in C++.

So in a data declaration * means pointer and & means reference. References and pointers are not the same thing. For example, references are not recursive. You cannot take the reference to a reference.

How do I use these data types? Well, in a statement (not a data declaration) & means “take the address of”. So I can write:

```cpp
Penguin pete;
Penguin* p_ptr = &pete;
```

So if I do this, p_ptr becomes a variable of type “pointer to Penguin”. Its value is the virtual memory address of pete. There is no abstraction here. It is a raw virtual memory address.

Why would I want to do this? In the example above, I took the address of a local variable. There are times to do this, but it’s not common. The more common way to get a pointer is through the new operator:

```cpp
Penguin* p_ptr = new Penguin();
```

The new operator allocates a chunk of memory on the stack (of the right size for the declared data type) and invokes the constructor.

For those of you who have programmed in straight C, new replaces malloc. In fact, it is implemented in terms of malloc. However, you should never call malloc directly. Only use new. Mixing malloc and new can have bad effects.

I can also go the other way. In a statement, * means go to the address of this pointer. So to call the Size() method of my penguin in the stack I write:

```cpp
(*p_ptr).Size();
```

Because *p_ptr is the penguin pointed to by p_ptr. This is cumbersome, however, so I can write

```cpp
p_ptr->Size();
```

The arrow operator says “follow the pointer, and then go to the field or method pointed at”.