CS225 : Data Structures and Software Engineering
Pointers and Memory

Jason Zych

Chapter 1

Pointers and Memory

1.1 The layout of real memory

Part of the goal of high-level programming languages is to hide the low-level details from you. The first programmers had to write programs using only 0's and 1's, and directly manipulated the values stored in hardware. But, over the years, we have created better and better languages with with to express instructions to the computer (and, of course, along with those languages, we had to create the tools – compilers – to translate those languages to the 0's and 1's which were still the only thing that the hardware could understand). These days, there is often no need to directly manipulate hardware. Our programming languages allow us to think in terms of "variables", and recently, "objects". Yes, we know that behind the scenes, data is being stored in memory, in the form of strings of bits, but we prefer to think in terms of variables and objects when writing software, and would generally like to ignore the idea of real memory entirely, in favor of abstractions that are more closely tied in to the problems our software is supposed to solve.

However, there *are* times when having knowledge of, and access to, real memory could be helpful, primarily because by directly manipulating memory, we can sometimes make our programs faster or make them take up less memory overall. The ability to handle those times is one of the things that distinguishes C++ from Java – in C++, we *can* directly manipulate memory. We don't *have* to...but we can if we want to.

So, let's move beyond our standard abstractions and take a look at actual memory for a moment. You will learn more about the particular hardware details in courses such as CS231 and CS232, and you will learn more about the details of compilation in CS326 if you get a chance to take that course. What follows is certainly a simplified description of how things really are, and the vast majority of the information in these first two sections is **NOT** something you need to know for this course. We present it here briefly because having a basic understanding of memory and compilation will help you to better understand the memory manipulation topics we are about to discuss. If you want, you can skip ahead to section 2.3 and refer back to this section or section 2.2 only if you get confused or want to know a bit more information.

The collection of memory in a computer can be thought of as an array of "cells". Each cell contains a collection of bits (in today's machines, a typical cell is 32 bits long) that together represent some kind of data. Each cell also has an address. This is why the array analogy is appropriate. Just as a Java array is a collection of individually numbered cells (see Figure 2.1), the memory in a machine is also a collection of individually numbered cells (see Figure 2.2).

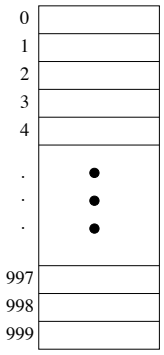

Figure 1.1: A 1000-cell array in Java (or most other languages)

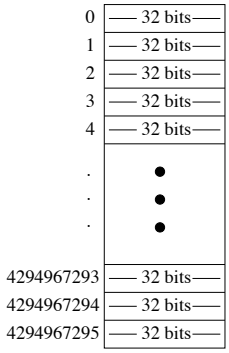

Figure 1.2: Physical memory, which strongly resembles an array. The collection of available memory can be very large – the above memory "array" contains 2^{32} cells

As you may remember from Java, some data types are less than 32 bits long, and so there are times we want to access halves or fourths of a cell to get 8 or 16 bit memory pieces. For that reason, we would actually also number the individual 8-bit pieces of a 32-bit cell, meaning the numbers assigned to the start of 32-bit cells would be multiples of four. (See Figure 2.3.) Why not just have 8-bit cells? Well, that's an issue better left for CS232. For this discussion, we will only use full 32-bit cells, but the above reasoning explains why our cell numbers will be multiples of four instead of multiples of 1.

In an array, the individual numbers that refer to particular cells are called *indices*, but when dealing with memory, the numbers that refer to particular cells are called *addresses*. So, in Figure 2.3, the addresses of the cells are 0, 4, 8, 12, 16, etc.

In a real machine, of course, indices such as '4", "12", or "28" are meaningless; everything is handled with 0's and 1's. That's okay, though, because the hardware used to access memory can simply represent values like "4" or "28" with binary numbers. The memory array is therefore really indexed with sequential binary numbers (see Figure 2.4).

For simplification's sake, we will use the notation **a4** or **a28** to refer to the bit patterns representing addresses (see Figure 2.5). This makes our discussion easier, but keep in mind that when we use an expression such as **a4** or **a28**, we really mean some pattern of 32 bits whose numerical value is 4 or 28, respectively.

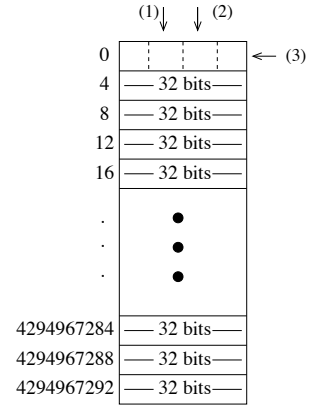

Figure 1.3: Memory that provides the ability to access 8-bit segments

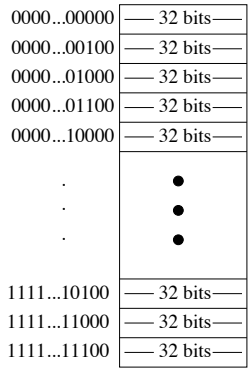

Figure 1.4: Memory which is addressed via binary numbers, as in real life. Again, cell addresses are multiples of four.

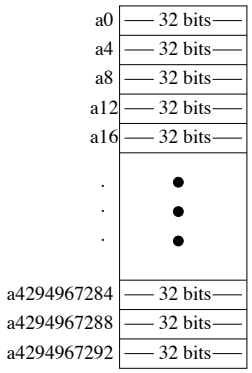

Figure 1.5: Our final "simplified" memory picture that we will use for the remainder of this packet – and for most of our memory discussions throughout the semester

How the memory array is then accessed – and accessed quickly – is beyond the scope of this course, but the simplified idea is that when the processor wants to use a particular location in memory, it sends a collection of 32 bits over to the memory unit (see Figure 2.6). This bit sequence is the address of some cell in memory. The memory unit compares that bit sequence to the bit sequences which index its memory cells, and when it finds the match, that is the cell that the processor wants (see Figure 2.7). The final step then depends on what the processor wants to do with this cell. If it wants to read the value, then the 32 bits in this cell are sent back to the processor (see Figure 2.8). If the processor wants to write to this cell, then it will also have sent, along with the address bits, a collection of 32 bits to write into the cell, and those bits will then be written into the cell, thus erasing the bits that *used* to be there.

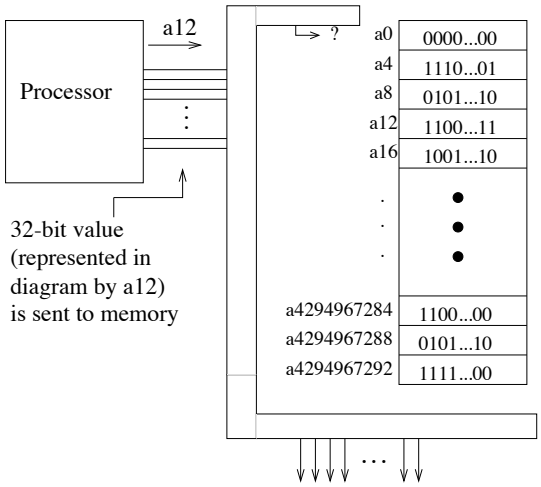

Figure 1.6: Using the memory unit, part 1: The processor sends an address to the memory unit

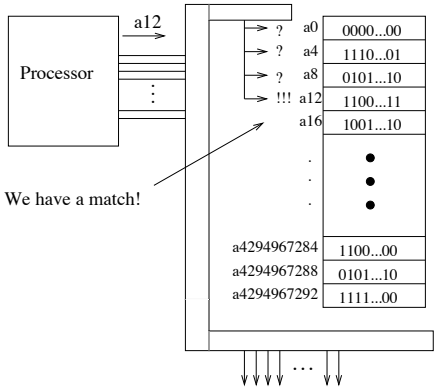

Figure 1.7: Using the memory unit, part 2: The memory unit finds which cell address matches the address it was sent

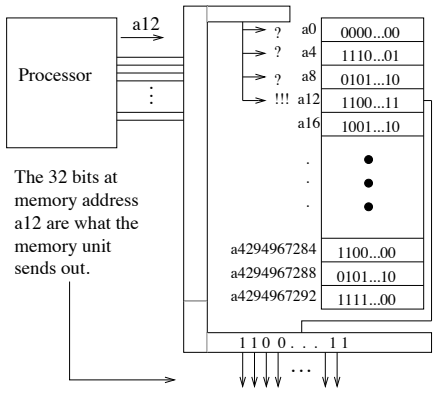

Figure 1.8: Using the memory unit, part 3: Once a matching address is found, the value in the cell at that address is sent back to the processor

1.2 Compiling a program

Since the programmer uses things like variable names and numerical values when writing programs, and the machine uses strings of bits, the compiler needs to handle the accurate translation from one to the other. When the compiler creates the machine language instructions that will eventually run on the processor, it has to create those instructions in such a way that the programmer's *requests* – for memory allocation, array access, assigning a value to a variable, or whatever – are correctly handled by the processor even though the processor has no way of understanding the variable name `n` or the symbol "5".

Consider, for example, the following snippet of code:

```
int n;
n = 5;
```

As you know, the first line is a variable declaration, which reserves some "space" for an integer value and calls that "space" `n`. On the machine level, though, the abstract notion of "space" and the idea of "naming" a "space" (with the name `n`) do not apply. Instead, the only ability the processor has in this regard is to write bit patterns into memory locations which are addressed by *other* bit patterns.

If that is the case, then how does a program actually work? How does the compiler manage to convey *your* ideas in the language of *bit patterns*? Well, the compiler needs to keep a large table of all the variable names you are using. When you declare a variable – for example, when the line `int n;` above is reached – the compiler enters the variable `n` into its variable table. At the same time, a memory location is chosen for the new variable. So, for example, if the compiler decided to store the value of `n` in the cell at address **a48**, then what would appear in the table would be the pair:

```
(n, a48)
```

Then, any other time you used `n` in your program, the compiler would look up `n` in its table, would see that it had decided to store `n` in the memory cell with address **a48**, and thus when it began writing machine instructions, it would use **a48** instead of `n`.

The same goes for any other declaration. If you had three more declarations,

```
int x, y;
float z;
```

then your compiler table might decide to allocate these three variables in the next three memory locations after `n`, meaning your compiler table would look as follows:

```
(n, a48)
(x, a52)
(y, a56)
(z, a60)
```

The point here is that, while the program you write can be filled with occurences of the variable name `n`, the machine code generated by the compiler will instead be filled with reads and writes to the memory cell with address **a48**. Everytime the compiler sees `n`, it can write **a48** to the machine code file. And, the compiler can do this because it keeps that internal table so that it can look up which variables correspond to which memory addresses.

This of course means that you, the programmer, don't need to worry about assigning memory addresses. You can just use abstract concepts – the ideas of variables such as `n` or `x` – and the compiler deals with translating those ideas to actual memory locations. And, it also means that the processor and memory unit don't need to know "variable names", since by the time the program is ready to run on the processor, the compiler has already translated all of your variable names to actual machine addresses.

This also means that your "variable names" cease to exist after compilation is finished. As long as the compiler is running – as long as it is in the process of translating your code to machine code – it needs to keep that internal table intact, so that it can look up variable names and retrieve their corresponding memory addresses. But, once the machine code is completed, there is no need for that "variable name" information anymore, because all the appropriate memory address information has already been written into the machine code file. So, the compiler deletes its table as part of the conclusion of the compilation processes. And, of course, if you decide to recompile, the table must then be reconstructed when the compilation process is started again.

This "variable table" can hold more than just pairs, though. In fact, one other very important piece of information it needs to hold is the *type* of the value. So, imagine our above table of pairs as a table of triples instead, where the type is stored along with the variable name and machine address.

```
(n, a48, int)
(x, a52, int)
(y, a56, int)
(z, a60, float)
```

Why is this type information important? Well, consider, for example, the 32-bit pattern

```
0100 0001 0110 1000 0000 0000 0000 0000
```

Imagine you have a variable `someVar`, and its corresponding address is **a324**, giving us the following variable table:

```
(n, a48, int)
(x, a52, int)
(y, a56, int)
(z, a60, float)
(someVar, a324, ????)
```

Notice we have hidden the type value. Now, imagine that the 32-bit pattern above is stored in the memory cell at address **a324** – that is, the variable `someVar` stores the bit pattern above. What will the following statement (a print command in Java):

```
System.out.println(someVar);
```

print to the screen? Well, you know the bit sequence in the cell at address **a324** is the value you are dealing with, but in spite of that, you can't fully answer the question because you don't know what the type of `someVar` is. If I told you that type, however, then you could correctly interpret the bit pattern. For example, if I told you that the type of `someVar` – and hence the type of the bit pattern – was the type `int`, then you could realize that the bit pattern represented the number 1,097,334,784. If you happened to know the standard for representing floating point numbers in collections of 32 bits (a CS232 topic), and I told you that `someVar` was of type `float`, you could tell me that the bit pattern represented the floating-point number 14.5. But you can't tell me the answer until I tell you the type.

Likewise, the machine has no idea what value the bit pattern represents unless the compiler tells it the needed type information. So, when the above output statement is translated to machine code, the instructions the machine is given are not just, "go to machine address **a324** and retrieve that value for printing", but in addition, the machine is also given instructions explaining how to convert that value to a readable form. In other words, when the compiler tries to translate the above output statement to something the machine can understand, the output statement is translated to one set of instructions if `someVar` is an `int` variable, and the output statement is translated to a different set of instructions if `someVar` is a `float` variable. The first set of instructions say "retrieve the value at machine address **a324**, and interpret and print it via this procedure: <more instructions > so that the user views it as an integer". The second set of instructions say, "retrieve the value at machine address **324**, and interpret and print it via this other procedure <more instructions, but different than the ones in the first set> so that the user views it as a floating point number".

The compiler can only properly choose between the two sets of instructions if it knows what kind of value is stored at **a324**. That is, it needs to know what kind of value is stored in the variable `someVar`, i.e. it needs to know the type of `someVar`. But, as long as the compiler knows the correct type, it can give the machine the proper instructions to handle the bit pattern, which is why the machine *doesn't care* what "type" the bit pattern is.

Read that last phrase again: the machine doesn't care what type the bit pattern is. That phrase is very important, because it means that for any type we can possibly conceive of, as long as we can create a one-to-one mapping between values of that type and long strings of bits, and as long as we can write a compiler that can understand that mapping and convert it to machine instructions, we can represent that type in a computer. This applies not just to printing values, but to simply using values in general (for multiplication, or initialization, or whatever), since we ultimately need to convert them to bit patterns in order for the machine to work with them.

For example, the type `double`, as you may recall from Java, requires 64 bits of space. There is a standard for converting double-precision floating-point numbers to patterns of 64 bits, and converting patterns of 64 bits back to double-precision floating point numbers (again, to be seen in CS232!). And, the Java compiler understands this conversion and can make use of it. This means we can use the type `double` in our programs.

The same goes for any user-defined types in Java. Some user-defined types are only composed of member data of built-in types, so converting those user-defined types to bit patterns and back is simply a matter of converting the member data one by one. Other user-defined types have member data whose types are *also* user defined types, but now you just have extra levels of conversion – you convert *this* user-defined type to a bit pattern by converting *its* user-defined member data to bit patterns, which you do by converting *their* member data to bit patterns, and so on, getting more detailed until you are converting built-in types to bit patterns, something the compiler knows how to do. Since all user-defined types are ultimately just collections – possibly vast collections – of built-in types, you can use user-defined types in basically the same ways that you use built-in types. This is why your own classes work with the existing language as easily as they do.

So, that is how memory is set up in a machine, and how it is used as your program is compiled and executed. This discussion is relevant because it is language-independent: though the types and conversions may change, ultimately the same process goes on at the machine level regardless of what language you are programming in. So, this means that when we program in C++, the same kinds of things are happening inside the real machine.

Now, again, the details of these first two sections are not things you need to know for this course. But, hopefully, having seen some of those details will make the following discussion a little easier to understand, because some of the questions you might have otherwise had have now been answered in advance. As long as you don't talk about real memory at all – as you can do when learning Java – you can get by just fine with abstractions such as variable names. However, the moment you begin talking about real memory – as we will have to do when discussing C++ – you will naturally start to wonder how various language features are handled in real memory. These first two sections have been our attempt to give you a brief introduction to how some of the abstract ideas you learned in CS125 are handled "behind the scenes" in an actual machine and by a actual compiler.

The ultimate goal of all this is to explain a single type in C++ that doesn't appear in Java, a type that gives us the knowledge of real memory and the access to real memory that we spoke of at the start of this section – a type known as a *pointer*. We will be discussing pointers shortly, but first there are some more basic things to go over. The following section is where the "pointers and memory" lecture actually begins.

1.3 Java allocations versus C++ allocations

In Java, built-in types and user-defined types had separate allocation rules:

1. When you declared a variable of a built-in type (i.e. primitive type), via a statement such as `int n;`, memory was set aside to hold an integer value, and you accessed that memory by using the variable name `n`.

2. When you declared a variable of a user-defined type, via a statement such as `Coord c1;`, that merely created a reference, and you *still* didn't actually have an object of a user-

defined type – i.e. unlike built-in types, variables of user-defined types did not actually contain the space to store the data needed by the user-defined type. To get that space, you needed to *allocate* an *object*, via the statement `c1 = new Coord();`. That statement would allocate a new `Coord` object and set `c1` to refer to it. Variables of user-defined types contained only references to objects, not the objects themselves, and the objects had to be allocated off a section of memory called the *heap*. (If you have forgotten a bit of this, go back and reread pages 122 through 125 of your CS125 textbook.)

Note that this gives us two options – we can declare variables of built-in types, or allocate objects of user-defined types. We are pointing this out for the purposes of contrasting it with C++. In C++, you have *four* options:

1. You can declare variables of built-in types, just as in Java
2. You can allocate objects of user-defined types, via expressions such as `new Coord();`, just as in Java
3. You can declare variables of *user-defined* types, which you cannot do in Java. That is, in C++, you can make a declaration such as `Coord c1;`, and this declaration does *not* create a reference. Instead, it sets aside enough memory to hold all the data a `Coord` object needs, and then you can go ahead and manipulate that data via the name `c1`, without having to perform any allocation using `new`.

   This raises an interesting terminology question, in that this appears to be a variable (it has a name and is created with a declaration), but is also an object (it has the space needed for a user-defined type, and the values stored in that space will be values of a user-defined type). For now, just accept that the *terminology* will get a bit murky, and focus on on the *ideas*.
4. You can allocate "objects" of built-in types, something else you cannot do in Java. Again, the terminology doesn't quite match what you are used to, which is why we put the word "objects" in quotes. But, the idea is that, just as you can use `new` to allocate the necessary space for an object of a user-defined type, you can also use `new` to allocate the necessary space for an "object" of a built-in type. The result would be expressions such as `new int`.

So, in C++, we have four separate options for generating pieces of memory for our use. Two of them are familiar from Java (1 and 2) and two are not (3 and 4). Two of them involve built-in types (1 and 4) and two involve user-defined types (2 and 3). And, most importantly for our next topics, two of them involve simple declarations (1 and 3), and two of them involve the use of `new` (2 and 4). What we will do now is examine real memory and how the above four options are handled in real memory. We will deal with options 1 and 3 first, since those are both declarations of variables and are thus related. After that, we will explore the C++ type known as a *pointer*, which has some similarity to the idea of "references" in Java. Finally, will discuss allocation options 2 and 4, which both use `new` and which both make use of something known as *dynamic memory*.

1.4 Variables in real memory

Now that we have discussed the allocation possibilities in C++, let's examine how variables in C++ are stored inside physical memory. We will start with the memory layout in Figure 2.9,

where the rectangles are memory cells and the **a44**, **a48**,... markings to the left side represent the memory addresses of the cells, i.e. the specific bit patterns used to access those specific memory cells in the physical memory of the machine. (For more details, see section 2.1.)

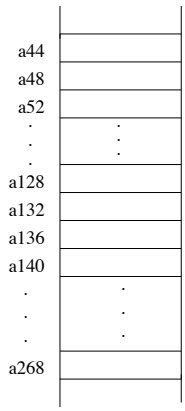

Figure 1.9: Our initial memory layout

Now, let's consider a simple variable declaration:

```
int n;
```

The purpose of a declaration such as this is to set aside some "space" and name that "space" with the variable name "n". What really happens is that a cell is set aside in memory for this new integer value, and the compiler keeps track of which memory cell we are talking about when we refer to `n`.

So, let's assume that the memory cell set aside to store the value of this particular variable is the memory cell with address **a44**. There was no particular reason that the variable `n` *had* to be located at **a44**; that's just where we put it in this particular example. It could just as well have been located at **a48** or **a268** or wherever else you might prefer. (Actually, there are specific ways in which that choice is made when a real program runs, but we won't be getting into that in this course – instead we'll just pick convenient locations and leave the discussion of the actual location selection to CS326.) Figure 2.10 shows the results of this declaration in our memory array.

Note that the variable `n`, and indeed, *all* variables, have the following properties:

1. A variable has a *memory address*, i.e., *where* the variable is stored.

2. A variable has a *value*, i.e., *what* is stored in the memory location.

3. Finally, a variable has a *name*, i.e. *how* this variable is referred to.

As you can see from Figure 2.10, our convention will be to write the name of a variable to the right of the cell the variable is located in, and to connect the name to the cell with a dotted line.

In our example above, the variable has the name `n`, and its memory address is **a44**. Note that there is no value yet – we have not assigned one to the variable. This raises another convention

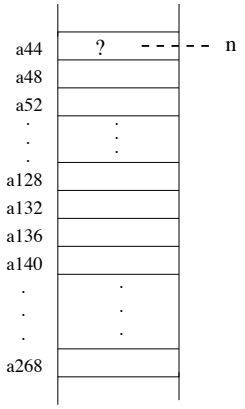

Figure 1.10: Memory with `n` declared

we will use, and also an interesting point. In Java, all built-in types (by which we mean only the primitive types – `int`, `float`, etc – and not the library types) are automatically initialized to default values when you declare them. In C++, they are not, so if you have not initialized a variable, then it has some meaningless, "garbage" data stored inside it (because there has to be *some* combination of electrons or lack of electrons inside the actual hardware). If we know that the value of a variable is garbage data, we will notate it with one or more question marks (as seen in Figure 2.10).

If after the declaration, we execute the assignment statement:

```
n = 5;
```

then now our memory looks as it does in Figure 2.11, and our variable has value 5 in addition to name `n` and memory address **a44**.

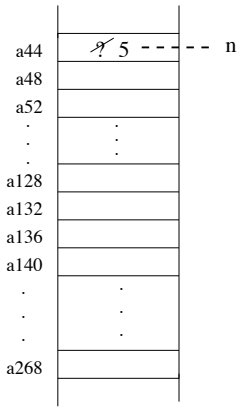

Figure 1.11: `n` is assigned the value `5`

We could summarize this variable as follows, if we want:

```
Current status of the new integer variable:
   WHERE it is located: a44
   WHAT it now holds: 5
   HOW we refer to it: n
```

The main idea here is that in both Java and C++ you have the ability to declare integer variables – variables of type `int` – and assign values to them. Inside memory, the scene appears as in Figure 2.11. Upon declaration of an integer variable, a specific cell is set aside for an integer value, and that cell is assigned the variable name that you selected (this "assigning of a name" is done by the compiler – see section 2.2).

Where C++ *differs* from Java is that if you make an ordinary declaration of a variable of a user-defined type:

```
Coord c1;
```

then in C++ this produces an object. In Java, `c1` would only be a reference of type `Coord`, and you would still need to use `new` to allocate an object for the reference to refer to. But in C++, the above decaration sets aside memory cells for the entire `Coord` object, *not* just a reference. There is no need to use `new`; the object is already allocated, and it is named `c1`.

But, if `c1` takes up space for an object, and not just a reference, the natural question is, how much space does it take up? When we declared an integer variable, it took up 32 bits – one cell. How many cells does an object of class `Coord` take up?

Well, behind the scenes, an object simply needs space for its member data. The class `Coord` had two member variables, and each was of type `double`. Since `double` takes up 64 bits – twice the size of an integer – and since we have two of them, we expect that an object of type `Coord` needs 4 cells, two to represent the first `double`, and two to represent the second `double`. The result of this variable declaration is shown in Figure 2.12.

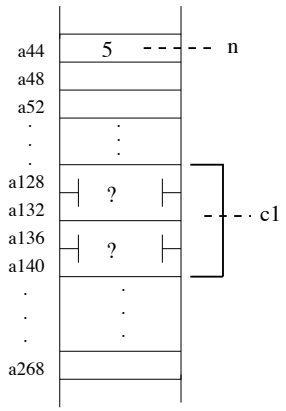

Figure 1.12: Memory after `c1` is declared. The first two cells work as a 64-bit unit to store the first `double`, and the second two cells likewise work together to store the second `double`.

We know from both our declaration and from Figure 2.12 that the name of the variable taking up the four cells **a128**, **a132**, **a136**, and **a140** is `c1`. And, since we have not initialized

this variable yet, we know it has a garbage value. However, it seems a little unclear what the memory address is. After all, `c1` takes up four cells, **a128**, **a132**, **a136**, and **a140**. Are they collectively the memory address? Can an object have four (or more) different addresses? Or, if only one is the memory address, which one is it?

Well, it *is* important to be able to refer to an object with a single address. Yes, the object takes up many cells – four in the above case. However, when we talk about the "memory address" of an object, we generally want to be referring to just one address, and so the convention is to have the first address be the "official" address of the entire object itself. That is, the address of the starting cell of the object is also the address of the entire object. In the above case, this is **a128**, and so the memory address of `c1` is **a128**. Whenever we have an object anywhere in memory, the address of that object is the address of the first cell given to that object.

We still need to assign a meaningful value to this variable. Right now, `c1` is not initialized, and so it has a garbage value. To fix this, we need to call the member function `Initialize()`, which as you recall will take two values as parameters and set the internal member data to be equal to those parameter values. This member function can be called from the object in exactly the same way as it is done in Java, namely by using dot notation. Simply list the object, followed by a dot, followed by the member function you wish to call.

```
c1.Initialize(4.1, 5.6);
```

Now, the values 4.1 and 5.6 are stored in memory, and so our status for our `Coord` variable is as follows:

```
Current status of the new Coord variable:
   WHERE it is located:  a128
   WHAT it now holds:    4.1, 5.6
   HOW we refer to it:   c1
```

The result can be seen in Figure 2.13.

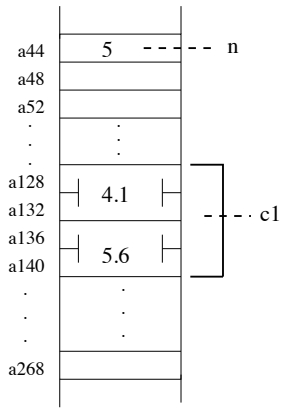

Figure 1.13: Memory after `c1` is initialized

So, aside from the key difference that `c1` is an object and not a reference, this is not really any different than Java. You can declare variables of type `int`, and in doing so you create "space" for `int` values and can read them and assign `int` values to those "spaces". And, likewise, you

can declare varaiables of type `Coord`, and in doing so you create `Coord` objects, and can operate on them in whatever way the `Coord` member functions allow. For `int`, of course, the functions (such as $+$, $*$, and $-$) are built into the language. For `Coord`, the member functions are invoked off the object name using dot notation, just as in Java. The only difference is that the variable itself is the `Coord` object, rather than just being a reference to the object. Other than that, the ideas are the same and your usage of the variables is not any different in C++ than it was in Java.

1.5 Pointers

Now, we start to approach the other half of things. We've seen how in C++, `int` variables are allocated exactly as they are in Java, and we've seen how, unlike in Java, we obtain `Coord` objects, and not `Coord` references, when we declare `Coord` variables in C++. The other half of the issue is that, in C++, it is also possible to have "references" to `Coord` objects (just as in Java), *and* to have "references" to `int` objects, which is something you could not do in Java.

C++ does indeed have something called a reference, but that is not the same as the "reference" we referred to in the previous paragraph. In this context, the feature in C++ that best compares to a reference in Java is a feature known as a *pointer*. In Java, you didn't have a type called `reference`; rather, each user-defined type (or library type, i.e. non-primitive type) in the language could be referred to by a reference of its own type. If you made the declaration `Coord c1;`, you created a `Coord` reference that could then be used to refer to an object that you allocated using `new`. But, that had to be a `Coord` object. That is, the statement `Coord c1 = new Coord();` was acceptable, but even if you happened to have defined a class `Coord2` as well, the statement `Coord c1 = new Coord2();` would not be acceptable. A `Coord` reference could refer to a `Coord` object, but not to a `Coord2` object.

So, in a sense, each user-defined type in the language had its *own* corresponding reference type. (Actually, since Java has an extensive built-in library, keep in mind that everything we say for "user-defined" types also applies to all of the library types.) Likewise, in C++, there is a pointer type corresponding to each different type in the language. But, just as all Java references basically behave the same way, in C++ all pointers basically behave the same way. The only difference between two Java references is the type each reference is associated with, and likewise, the only difference between two C++ pointers is the type each pointer is associated with.

Below is an example of the syntax for declaring a pointer variable:

```
int* numPtr;
```

Like any other variable declaration, in the declaration above we simply have a type followed by a name followed by a semicolon. However, note that this is not a declaration of an `int` variable, due to the asterisk in the declaration. The type in the above declaration is not `int`, i.e. "integer", but rather `int*`, i.e. "integer pointer", or alternatively, "pointer to integer". The `int` and the asterisk *together* are the name of the type. (You can see the results of this declaration in Figure 2.14.)

The variable `numPtr` is a pointer variable, just as `n` is an integer variable. And, like all variables, it has a location in memory, it has a name, and it has a value.

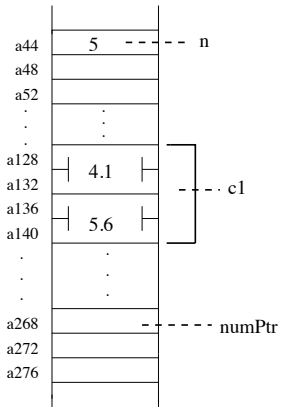

Figure 1.14: Memory after `numPtr` is declared

```
Current status of the new pointer variable:
   WHERE it is located: a268
   WHAT now holds:   ???
   HOW we refer to it: numPtr
```

The special feature of a pointer is the type of value it holds. Variables of type `int`, of course, hold integer values. Variables of type `float` hold floating-point values. Variables of type `char` hold characters. What pointers hold are *memory addresses*. That is, the bit patterns held in memory for pointer variables are not interpreted to be integers, or floating-point values (see section 2.2), but instead are interpreted to correspond to the bit patterns we use to address other cells in memory! For example, after the declaration above, we could have the statement `numPtr = a48`[1]. This statement writes the memory address **a48** (that is, the pattern of bits that you would use to access cell 48 of the memory array) into the cell named by the variable name `numPtr` (See Figure 2.15).

```
Current status of the new pointer variable:
   WHERE it is located: a268
   WHAT it now holds:  a48
   HOW we refer to it: numPtr
```

The important idea to start off with here is that, from a storage point of view, pointer variables are really no different than variables of any other type. A pointer variable still has:

[1]Please note that this would not actually be a legal C++ statement, simply because the expression **a48** does not represent an address in C++. We are using expressions such as **a48** or **a60** in these notes simply as a notational convenience, but in reality address values are printed out in *hexadecimal notation* (ex.: 0xeffff9e8), and hard-coding addresses into assignment statements requires casting integer values to addresses (ex.: `intPtr = (int *) 5;`). However, there is generally no reason to hard-code addresses into assignments, and we will certainly not be doing so in this course. We are just doing it momentarily at this point in the notes, in order to gracefully lead into the types of expressions we will *really* be using.

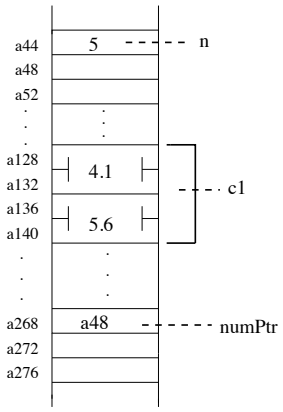

Figure 1.15: Memory after `numPtr` is initialized

1. a memory address – the above pointer variable is located at cell **a268**
2. a value – the above pointer variable now holds the value **a48**
3. and a name – the above pointer variable is called `numPtr`.

Pointers just happen to hold values of a different type than you are used to. Instead of integer values or floating point values or collections of such values (as with the class `Coord`), pointers hold addresses of other memory cells. That's the only real difference. When you read the bit pattern

```
0100 0001 0110 1000 0000 0000 0000 0000
```

in a memory cell represented by a pointer variable, you don't read it and think "this is the integer 1,097,334,784", and you don't read it and think, "this is the floating point value 14.5"; instead, you read it and think, "this is the address **a1097334784**, which is the address of a memory cell far downward near the bottom of the array."

So, if that's the only real difference, why make a big deal about pointers? Well, the reason pointers are useful is because of what can be done with their values. The values of other variable types – values such as 5, 4.2, or 'x' – are just numbers and characters that are relevant to *us*, but not particularly relevant to the *machine architecture*. However, a memory address is not just some value that the user is concerned with, but in addition it is an actual index into the physical memory array. For example, since `numPtr` holds the address **a48**, it is possible to read the value of `numPtr`, and once we have that value, **a48**, we can use it as an index into the physical memory array, *and retrieve the value located at the address* **a48***!*.

Since all pointers hold memory addresses, the values stored in pointers can be used as memory array indices to retrieve *other* values, just as in Figure 2.15, where the value in `numPtr` could be used to retrieve the value in cell **a48** even though `numPtr` itself was stored at **a268** which was nowhere near **a48**. The *values* of pointers are the *addresses* of other memory cells.

Well, so what? **a48** is empty. Yes, this is true, but `numPtr` didn't have to hold **a48**. What if we assigned it to instead hold the value **a44**? Now, the memory address that is the *value* of `numPtr` happens to be the same memory address that is the *address* of `n` (see Figure 2.16).

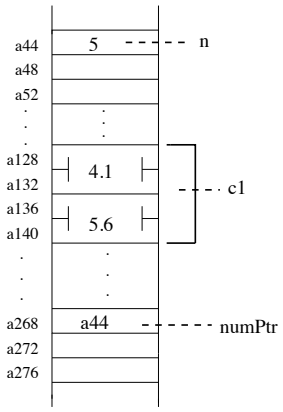

Figure 1.16: The value of `numPtr` is the address of `n`

Now, instead of using a (pseudocode) statement such as

```
print the value stored in n;
```

to print the value in `n`, you could use a (pseudocode) statement such as

```
print the value in the cell
 whose address is stored in numPtr
```

and the machine would go to **a268** (the cell indicated by `numPtr`, read the value there (**a44**), and jump to the cell addressed by that value (i.e. jump to the cell with address **a44**) and access *its* data. We would now be accessing the data in cell **a44**, without ever making use of the variable name `n` at all.

Now, there is no particular reason to avoid using the variable name `n`, but in general, it can be quite useful to access one piece of memory by storing its address elsewhere, as a value in some pointer variable. This is similar to Java, where it was helpful (and necessary!) to refer to objects by way of references to them.

But what we still need is an actual way of making the switch from "pointer variable that holds this address" to "memory cell that is located at the address which our pointer variable held as a value". Up to this point, we have simply said, "read the address stored in this pointer variable and jump to that cell". But how do we do this? What language syntax supports this idea? What we need is to have operations in the language that would handle this, operations that mean things like "go to the address that is the value of `numPtr`". And, in fact, C++ does indeed have two operations that allow this entire concept to work. These operations are:

1. The "&" operator, which means "address of", and
2. The "*" operator, which means "dereference"

Let's look at the "&" operator first. This symbol means different things in different contexts. The context we are concerned with here is the one in which the & symbol appears before a variable name on a line of executable code. (One example of a different context would be when the & symbol appears after a type name in the parameter list of a function header. We will

look at that context later.) When the & symbol appears before a variable name on a line of executable code, it means "address of". For example, the statement:

```
numPtr = &n;
```

is translated as "the variable `numPtr` is assigned as its value *the address of* the variable `n`." Since the address of `n` is **a44**, this statement is equivalent to

```
numPtr = a44;
```

The "address of" operator saves us from having to hardcode real memory addresses into the program. This is especially useful when it comes to typechecking in the compiler. The value **a44** by itself is just a memory address; *any* pointer could be assigned to hold it. But, if we use the address operator, then the compiler can make sure we are assigning the addresses of a type *only* to pointers to that type. That is, the assignment `numPtr = &n;` would be okay, since we are getting the address of an integer variable and storing it in a variable that was declared to be storage for integer addresses. However, if we tried to execute the statement `numPtr = &c1;`, then the compiler would realize that we are trying to assign the address of a `Coord` object to a variable designed to hold addresses of integers. Since this is wrong, the compiler could let us know we had made a mistake. It is much harder for the compiler to detect this mistake if we use a statement such as `numPtr = a128`.

The second operator used with pointers is the "*" operator. Again, this symbol means different things in different contexts – certainly at times it refers to multiplication! – but when placed before a pointer variable name on a line of executable code, it means "dereference", or more completely, "return the object at the address stored by this pointer variable" (where "object" here refers to any allocated memory chunk, whether for a user-defined type or for a built-in type). For example, if the variable `numPtr` holds the address **a44**, then the expression:

```
*numPtr
```

will return the object addressed by **a44**. So, the statement:

```
*numPtr = 6;
```

will first dereference the pointer variable `numPtr` to obtain the object at address **a44**, and then will assign the value `6` to that object. The end result is that, provided we have previously executed the statement `numPtr = &n` that we dicussed above, the following two statements:

```
*numPtr = 6;        AND     n = 6;
```

are *exactly* equivalent. The second writes `6` to the cell named `n`, which is the cell with address **a44**. The first writes `6` to the cell whose address is the value stored in `numPtr`, and that value is **a44**. The two statements mean the exact same thing (see Figure 2.17).

In fact, it should be clear that the & and * are inverse operations, meaning:

- `(*(&n))` is equivalent to `n`, and
- `(&(*numPtr))` is equivalent to `numPtr`

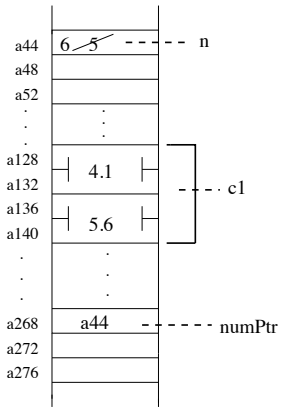

Figure 1.17: The command `*numPtr = 6;` first dereferences `numPtr`, which gives us the object located at `numPtr`'s value – i.e. the object at **a44**. Then, `6` is assigned to that cell at **a44**.

We can declare pointer variables of whatever type we want. The following code:

```
Coord* cAddr;
cAddr = &c1;
```

will set aside space in memory for a new pointer variable, and that pointer variable will be one which holds the addresses of `Coord` objects. Then, the code finishes after the second line, which assigns as the value of `cAddr` the address of `c1` (see Figure 2.18).

```
Current status of the new pointer variable:
   WHERE it is located: a276
   WHAT it now holds:  a128
   HOW we refer to it: cAddr
```

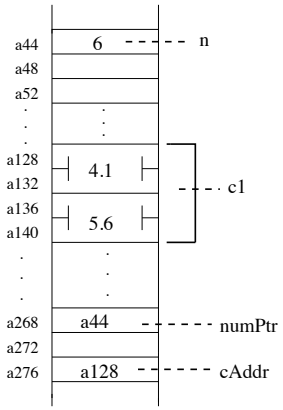

Figure 1.18: Pointer to objects of type `Coord`

1.6 Stack Memory

Our picture of memory so far is accurate, but incomplete. There are actually two different types of memory available to us, and up to this point, we have only dealt with one of them. In this section, we will give some more detail about this first type of memory, and in the next section, we will discuss the other type of memory. As with the earlier memory discussion (section 2.1), the discussion of memory here will be simplified somewhat, but it is accurate enough for our purposes.

On one end of the memory array, you have *stack* memory. This memory is also known as *local* memory, since it is the memory used to store local variables. The *stack* memory is so called because this is the memory used to implement the "function stack". In order to understand this, it is necessary to give a small introduction to how function calls are implemented in a computer.

Imagine that you are in your "start function", which in both Java and C++ is called `main()`. Furthermore, imagine that the first few lines of the function `main()` are as follows:

```
int main()
{
   int i, j;
   i = 20;
   FunctionA(i);
      .
      .
      .
```

(Functions in C++ can be global – i.e., standing alone rather than being a member of any particular class. That is why `FunctionA` can be called above without involving a class or object of some sort in that function call.)

Further, imagine the first few lines of `FunctionA` are as follows:

```
void FunctionA(int x)
{
   int k;
   FunctionB(x);
   FunctionC();
      .
      .
      .
```

And, perhaps the first few lines of `FunctionB` are as follows:

```
void FunctionB(int n)
{
      .
      .
      .
```

When you first start the program, space is set aside at the beginning of stack memory for the information of the function `main`. This "information" includes things like local variables, so the local variables `i` and `j` will be in this memory. This space in memory, where all the information

for a function is stored in one place, is known as a *stack frame* (see Figure 2.19). (We are not really going to get into the details of stack frames other than to just think of them as "boxes of information", and in fact I would rather not even have to use the actual technical term "stack frame", except that it is convenient to call these "boxes of information" *something*, and if we are going to call them something we may as well call them by their correct name.)

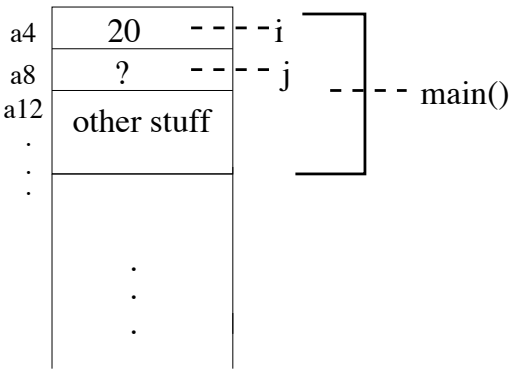

Figure 1.19: The stack frame for `main`

Now, we are currently in `main` and are about to jump to `FunctionA` via a function call, meaining we are leaving `main` and won't be returning to do any more work in `main` until we are finished with `FunctionA` and have returned from it. So, right where the stack frame for `main` leaves off, we can build another stack frame, this one for `FunctionA`. We don't need to worry about the stack frame for `main` now having no room to expand, because by the time we are ready to return to `main`, we will be finished with `FunctionA`, and so the stack frame we are creating for `FunctionA` could then be removed to make more room for `main`'s stack frame to expand. (More on that in a bit.) What do we store in the stack frame for `FunctionA`? Well, the first thing we store will be the parameters passed to `FunctionA`, but after that we would store local variables and other information for `FunctionA`, just as we did for `main` (see Figure 2.20).

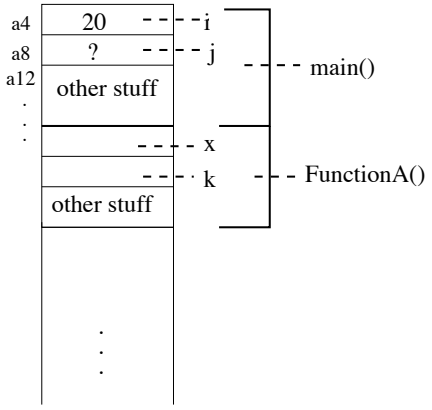

Figure 1.20: The stack frame for `FunctionA` is added

Finally, `FunctionA` calls `FunctionB`, which means that for the moment we are leaving

FunctionA and we know we won't be adding anymore information to FunctionA's stack frame. So, right where FunctionA's stack frame leaves off, we create a stack frame for FunctionB, which would contain *its* parameters, *its* local variables, etc. (see Figure 2.21).

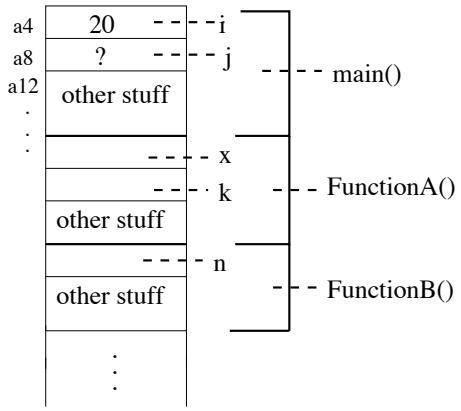

Figure 1.21: The stack frame for FunctionB is added

The idea is, as long as we are making function calls, we are building these stack frames for each function. This would be bad – we'd eventually run out of memory – except for the fact that eventually we erase these stack frames and reuse the memory. Specifically, we erase a stack frame when we return from its function. So, for example, when we return from FunctionB, it makes no sense to hold on to the information in FunctionB's stack frame. After all, FunctionB is done! So, when we return *from* FunctionB, we return *back to* FunctionA. And, likewise, at the same time we will erase FunctionB's stack frame, and return our attention to the end of FunctionA's stack frame (see Figure 2.22).

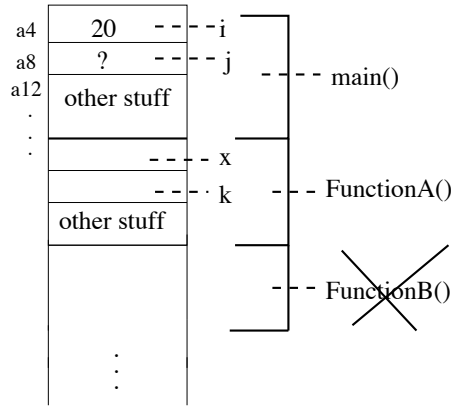

Figure 1.22: The completion of FunctionB results in the erasing of FunctionB's stack frame

When we then call FunctionC from FunctionA, the same thing applies. When we call FunctionC, a stack frame is created for FunctionC (Figure 2.23), and when we return from FunctionC, we erase FunctionC's stack frame and return our interest to FunctionA's stack frame (Figure 2.24). Finally, when we return from FunctionA back to main, the stack frame for

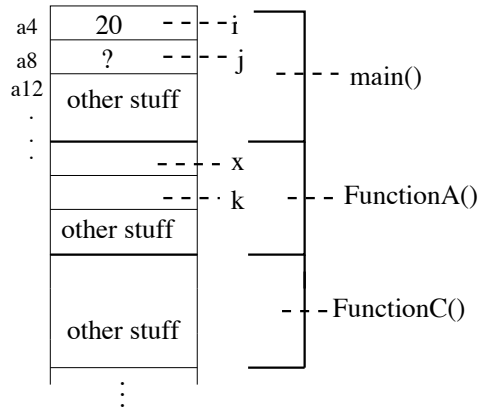

Figure 1.23: The stack frame for `FunctionC` is added, writing over the memory where the stack frame for `FunctionB` used to be.

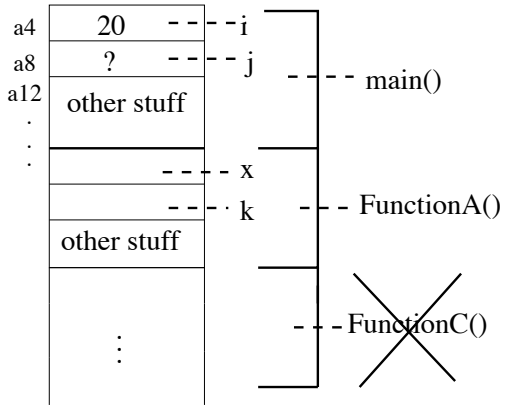

Figure 1.24: The completion of `FunctionC` results in the erasing of `FunctionC`'s stack frame

`FunctionA` is erased, and we are left with only the stack frame for `main` (Figure 2.25), which is finally erased when we exit the program.

That is the basic method behind how local information is stored and erased as function calls proceed – when you call a function, a new stack frame is created for it, and when you return from this function, that stack frame is deleted. Since it would be impossible to return "two levels back" (for example, you can't return to `main` from `FunctionC`, you have to go in reverse order – i.e. you have to go back to `FunctionA` and then from *there* return to `main`), you know that you will always build your stack frames one-by-one downward, and erase them in the reverse order.

Whenever you have a variable declaration in your code, that variable is allocated off the stack. That is, the memory cell for that new variable is located in the stack frame for the function that that variable is declared in. When we declared `int k;` in `FunctionA`, the memory for `k` was located in `FunctionA`'s stack frame. Therefore, *all* variable declarations are by definition, local variables. Some of them might have *very* long lives. For example, any local variable declared at the front of `main` is going to last for the life of the program, since it won't be erased until the stack frame for `main` is erased at the end of the program. But, ultimately, all variables are

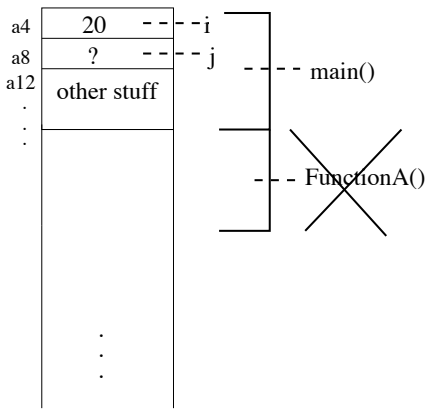

Figure 1.25: The completion of `FunctionA` results in the erasing of `FunctionA`'s stack frame

destroyed when the function they were declared in reaches its end.

As a result, objects that are local variables – for example, the object we create with the declaration `Coord c1;` – are often called *local objects* or *stack objects*. They are objects, but they are also local variables, and the memory they take up is memory that comes from the "stack memory" area. By the same reasoning, local variables of built-in types – such as the `int` variable `n` produced by the declaration `int n;` – could also be thought of as local "objects" or stack "objects". The terminology is blurred somewhat, due to the similarity between the way local variables of built-in types are handled, and the way local variables of user-defined types are handled.

Now, take note of an important detail – when you leave a function, all of its information is erased when the stack frame is erased. *That* is why local variables do not last beyond the function call – they are erased from their memory cells once the function is over. In fact, the stack frame for `FunctionC` was written to the same memory cells that were used to hold the stack frame for `FunctionB`! We can do this because the values in `FunctionB`'s stack frame are no longer accessible and no longer needed once `FunctionB` has ended, so there is no reason to keep that information untouched, and it is okay to write over it with new information that is relevant to the function we are *currently* in (as we do when we call `FunctionC`).

So, we can *count* on our local variables being destroyed when we exit a function. We *know* it will happen. Therefore, it is quite dangerous to try and use local variables after we have left their function. For example, imagine that our previous code for `FunctionA` and `FunctionB` really looked like this:

```
void FunctionA(int x)
{
   int* k;
   k = FunctionB(x);
   *k = 20;
   FunctionC();
   print *k;              // pseudocode
   // whatever other code might come next
}
```

```
int* FunctionB(int n)
{
   int temp = 30;
   // more code could be here
   return &temp;
}
```

In our previous picture, the stack frame for `main` started at **a4**, but the other addresses were not shown. Let's now imagine that the stack frame for `FunctionA` starts at **a64** and ends at **a100**, and the stack frame for `FunctionB` – now with a local variable – starts at **a104** and ends at **a132** (see Figure 2.26). Now, note that the return type for `FunctionB` was `int *`, and what we returned was the address of the local variable `temp` of `FunctionB`. Since we are returning the address of an integer, our return value matches our return type, which is "pointer to integer".

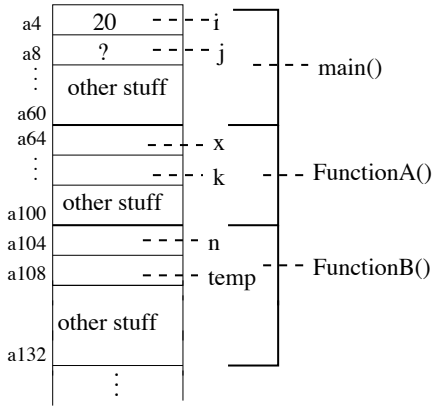

Figure 1.26: We are preparing to return the address of `temp`

Once `FunctionB` returns, `FunctionA` would be given the address of this local variable of `FunctionB`, and would store this address inside the variable `k` (see Figure 2.27). We could even alter the value of this local variable, because having pointers that hold addresses means we can directly manipulate the cells at those addresses.

But, what happens when we call `FunctionC`? Well, the stack frame for `FunctionC` gets set up, and the memory at address **a108** is claimed for `FunctionC`'s stack frame. And thus, when `FunctionC` returns, the no-longer-used value that is sitting at address **a108** is a value from `FunctionC`. Our value `20` is gone! So, when we try to execute the line `print *k;`, we do not print the value `20`, but instead we print whatever random value happens to be sitting in that cell as a result of its use by `FunctionC`. Our data has been corrupted! (See Figure 2.28.)

Well, no, we shouldn't exactly say that our data has been corrupted. It has been overwritten due to a natural course of events within the machine. Since the machine has every right to do this, *we* are the ones that are in the wrong. You should *never* return the address of a local variable as the return value of the function you are leaving. It is inherently unsafe, because it means you are giving the function to which you are returning an access to your local variables that it should not have. You are allowing it to read and write to memory that could very easily be written over by the next function call, and thus your program is relying on data that could very easily and quite naturally be corrupted by the next function call.

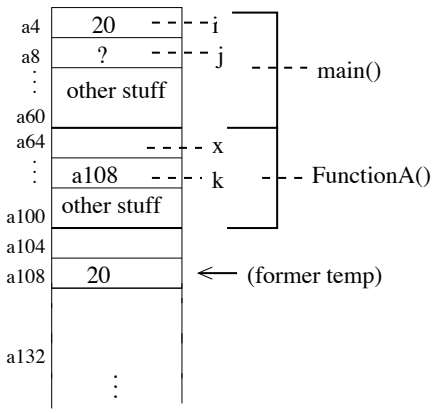

Figure 1.27: `FunctionA`'s local variable `k` now holds the address (**a108**) of the former "`temp`", a local variable whose scope has ended. `FunctionA` has assigned a value to the cell at **a108** via the statement `*k = 20;`, but that cell can be claimed by the very next function call, and thus the existence of the data that is stored there (the integer 20) could end at any time. This is dangerous!

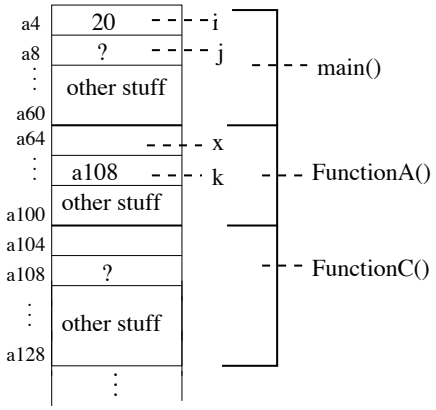

Figure 1.28: When the stack frame for `FunctionC` is created, any information that we were "lucky enough" to still have at address **a108** is now overwritten. If later, upon our return to `FunctionA`, we try to access **a108** through the pointer variable `k`, we will find that this memory cell no longer holds the integer 20 that we expected it to hold, because the stack frame of `FunctionC` wrote over that value. DO NOT pass back addresses of local variables, and DO NOT attempt to use memory cells from the stack once the stack frames they were in have ceased to exist.

Again – *never* return the address of a local variable. By the time you finish returning, the function you return to will *already* hold the address of garbage data – and indeed, it *will* be garbage data philosophically, even if you technically could mess with that memory cell for a bit before calling the next function and writing over the cell. *Do not* fall into the trap of thinking, "oh, I could be careful and not call a new function until I am done with this memory cell whose address was returned to me". If your program design requires that you play games like that, you have a poor design and you should rethink it. It is never worth it to play dangerous games like that and risk using data that could vanish at any time.

One more small and interesting note – due to some internal bookkeeping necessities (to be seen in CS232), every function call uses up at least a *bit* of memory, even if you have no local variables or parameters. This is why runaway recursion is so bad. If you don't have some way of ending your recursive function calls, you will allocate chunk after chunk of memory – one chunk for each function call and thus one chunk for each stack frame – moving downward and using up more and more memory until you finally reach the end and run out. You never have the opportunity to erase these chunks before reaching the end because you keep recursively calling functions without returning from any of them.

1.7 Dynamic Memory and `new`

There is also a second type of memory we can make use of, and it should be somewhat familiar to you because you dealt with in Java as well, even if you didn't discuss it in detail. This memory starts at the opposite end of the memory array, it is called *dynamic* memory, and is given to you whenever you use the command `new` in Java or C++. Cells allocated from this part of memory are not erased when you return from a function, because this memory is not local to any specific function. Rather, you can think of this part of memory as being a pile of cells from which you can pull a group of cells to store some data, mark those cells as your own, and have them remain marked as your own, storing your data, until *you* decide you are done with them. Because of this idea of a "pile" of cells, this section of memory is often called the *heap*[2].

Before we move on, we would like to note that the idea of "types of memory" is a completely artificial distinction. To use an analogy, imagine that you have bought a desk that you intend to put your books and homework papers on. The surface of the desk is basically the same from end to end, and you could put various books and homework papers wherever you want – there is nothing about the desk that, say, allows you to put a book in a particular spot but prevents you from putting a homework paper there. However, you would probably prefer to use some kind of organized system, rather than scattering things all over the desk. For example, you might put all your books on the left side of the desk, and all your homework papers on the right side of the desk. Or, you might do just the opposite. The point would be that you chose some method and stuck with it for the sake of organization, not because that was the only way you could put things on your desk. And likewise, all our memory cells are the same, but we find it convenient to have one side be the stack for function calls and local variables, and to have the other side be the heap from which to request memory cells using `new`. Using one side of the memory in one

[2]Incidentally, the word *heap* has two different meanings in computer science. The first refers to dynamic memory, and the second refers to a type of data structure implementation that we will discuss when we reach the topic of priority queues. It's just one of those quirks of history that the same term was used for two different ideas. It should generally be clear from context which definition of "heap" we are using – and of course, we haven't even gotten to the second definition yet, so that makes things easier still. :-)

way, and the other side of memory in the other way, makes memory far easier to use than if we had mixed both usages together and had stack frames mixed in with objects we created using `new`.

That said, let's examine how dynamic memory is actually used, by observing our memory diagram as some sample code is executed on the system. We will step through the following code, and take a look inside memory after each line is executed.

```
int* intPtrOne;
Coord *cPtrOne, *cPtrTwo; // the * must go before each variable
cPtrOne = new Coord();
intPtrOne = new int();
cPtrTwo = new Coord();
cPtrTwo->Initialize(2.3, 4.5);
cPtrOne = new Coord();
```

The first two lines of code are declarations of the type we already looked at in section 2.5, and so in our first memory diagram (Figure 2.29) we have already set side the space for those three variables. Notice that, since they are variable declarations, the memory we set aside for them comes from the stack, as we have discussed earlier.

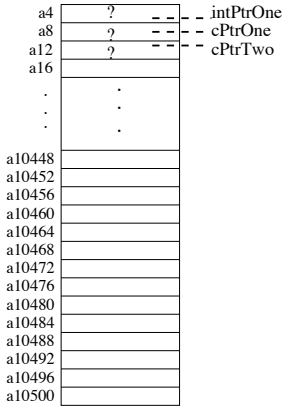

Figure 1.29: Our memory after the first two lines of sample code are executed

Now, on the third line, we are creating an object using `new`. Just as our variables were also called "stack objects" or "local objects", this object that we create using `new` is called a *dynamic object*, and we say that we are *allocating the object dynamically*. The expressions using `new` on lines 3, 4, 5, and 7 are all known as *dynamic* allocations. The word "dynamic" is used precisely because these objects can live beyond the ending of the function that they were first created in – that is how dynamic memory gets its name.

Figure 2.30 shows the result of this dynamic allocation. Four cells near the bottom of our memory are marked by the system as being in use, and together they compose the new dynamic `Coord` object. Of course, just as with the stack objects, this dynamic object is not initialized yet – we have set aside the memory for this object, and nothing more.

Now is a good time to formally discuss the keyword `new` in C++. In C++, `new` is a function. That is, when you execute the expression `new Coord()`, there is really a function `new` which is

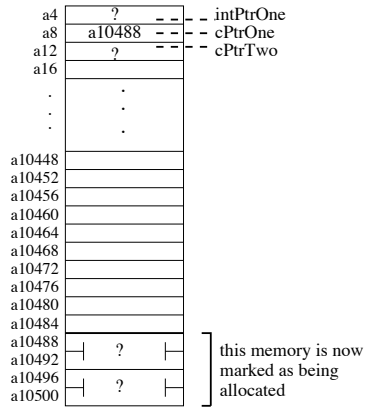

Figure 1.30: Our memory after the first dynamic allocation

being called. Without getting too much into the mechanics of it (since those are details you *really* don't need to know right now and they will only confuse the issue), the function `new` will perform the following tasks, in order:

1. Dynamically allocate enough memory to store an object of whatever type comes after `new`. In the above case, the type that comes after `new` is `Coord`, so `new` dynamically allocates enough memory to store a `Coord` object.

2. (ignore this step) Call the constructor indicated. For now, don't worry about this step – we will come back to it when we talk about constructors. (That is when we will talk about those parentheses after the type name, since they are related to the idea of constructors.) But, I at least wanted you to know that there *is* a second step here – even if we are going to ignore it for now – so that it won't come as a surprise later on.

3. Return the memory address of the newly allocated object. Right before this step, `new` has completed the work it needs to do with the memory it has dynamically allocated, and so its final action is to return a value – the address of the first cell in the group of cells that `new` has set aside for you.

So, our expression `new Coord();` sets aside the memory for a `Coord` object, does some other stuff we'll talk about later, and then returns the address of this new object. Of course, if we don't store that address anywhere, there will no longer be any way to retrieve the object – we will simply have this dynamic object existing in memory but we won't know exactly where. This is why we want to take that value returned by `new` – the address of our new object – and store it in a variable that is capable of storing an address. That is, we want to take the value returned by `new` and store it in a pointer. The statement `cPtrOne = new Coord();` does exactly that, and so when this statement has finished executing, not only do we have a newly-allocated dynamic object, but we have stored its address inside our `Coord` pointer variable, `cPtrOne` (as previously seen in Figure 2.30).

Line 4 works basically the same way, except that we are dynamically allocating an `int` "object", rather than a `Coord` object. The result is seen in Figure 2.31 – we allocate one cell for an integer value, and then store the address of this cell in the `int` pointer variable `intPtrOne`.

Likewise, Figure 2.32 shows the result of the execution of line 5, which allocates another dynamic `Coord` object and stores the address of this object in `cPtrTwo`.

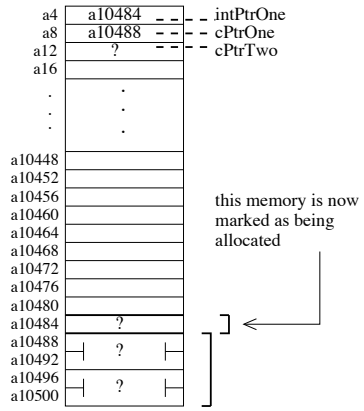

Figure 1.31: Our memory after the dynamic allocation of an `int` "object"

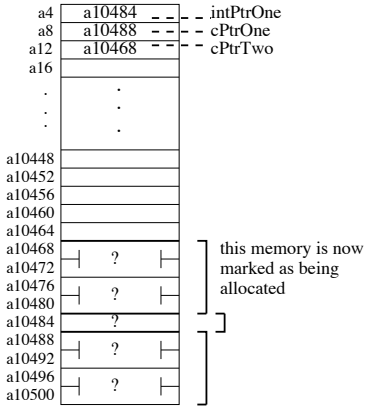

Figure 1.32: Our memory after the dynamic allocation of the second `Coord` object

Line 6 is a little more interesting, but not as bizarre as it first appears. In Java, you used references (created by declarations such as `Coord c1;`) to refer to objects, you used `new` to allocate the objects (via statements such as `c1 = new Coord();`), and member functions of an object were called by using the dot notation, resulting in statements such as `c1.Initialize(2.3, 4.5);`. In C++, a declaration such as `Coord c1;` produced the actual object, and not simply a reference to one. Therefore, there was no need for `new`. However, the syntax to invoke a member function remained the same; you could still invoke an object's member function using the statement `c1.Initialize(2.3, 4.5);`.

However, in C++, we can also have pointers to objects, and thus we would like a syntax for calling member functions by way of pointers to objects. We didn't need two syntax forms in Java, because we only had one way of calling a member function. We *had* to use a reference to an object – and thus the dot notation – because we could not assign names to actual objects, and thus *all* usages of a Java object – whether to call a member function of that object or to

do something else with that object – required that we access that object via a reference to it. Whereas in C++, in addition to accessing an object via a pointer to that object, we can also access an object directly, because we can assign variable names to objects. Java used the dot notation when using references to obtain objects, but we can not use the dot notation in C++ when using pointers to obtain objects, because we already use the dot notation in C++ when dealing directly with objects. So, what can we do?

The answer is that we need a new notation. This notation would take a pointer instead of an object, but would still allow us to call a specific member function. That is, the syntax we currently know in C++ is

```
(object).(member function)
```

What we need is a syntax that works as follows:

```
(pointer to object)syntax(member function)
```

And, the syntax used in C++ is the "arrow notation", i.e. the operator `->`. So, our syntax for calling a member function when given an pointer to an object is:

```
(pointer to object)->(member function)
```

So, just as the statement:

```
c1.Initialize(2.3, 4.5);
```

took an object and called its member function, the statement

```
cPtrTwo->Initialize(2.3, 4,5);
```

takes a pointer to an object and called that object's member function. Furthermore, since we can dereference pointers to get the objects whose addresses they hold, our statement above, with the arrow notation, is *exactly* equivalent to:

```
(*cPtrTwo).Initialize(2.3, 4.5);
```

So, the arrow syntax is merely a shorthand for "dereference this pointer to get an object, and then use the dot notation to call a member function of that object".

Therefore, in line 6 all we are doing is calling the `Initialize` member function of the object whose address is held by `cPtrTwo`, which is the object we allocated on line 5 using `new`. Since `Initialize` is simply going to initialize the object in question, once line 6 is executed, the object whose address is held by `cPtrTwo` now contains the values `2.3` and `4.5` instead of garbage data (see Figure 2.33). (Note that we could also have used the arrow notation back in section 2.5, when we were first assigning pointers to hold the addresses of local objects. The pointer doesn't *need* to hold the address of a *dynamic* object for this syntax to work; it can hold the address of a dynamic *or* a local object.)

Finally, in line 7, we are simply doing one more dynamic allocation, and this dynamic allocation works the same way the others so far have worked. The only quirk here is that we are not writing the address of the new object to a new pointer – instead, we are writing over the address currently stored in `cPtrOne`, which means that, even though we have a variable storing the address of the *newest* dynamically allocated object, we no longer have any way of accessing

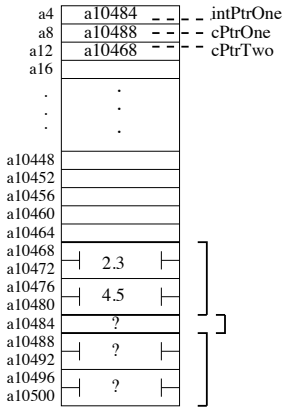

Figure 1.33: Our memory after the `Initialize` call

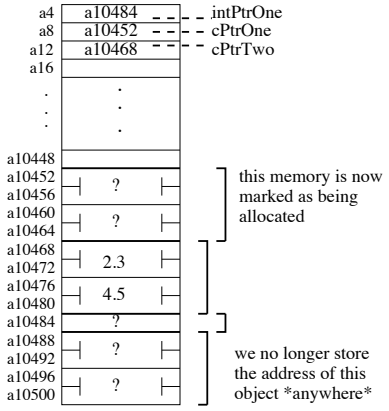

Figure 1.34: Our memory after the last dynamic allocation

our *original* dynamically allocated object, the one located at **a10488** (see Figure 2.34). Yes, it is still in memory, but we don't hold its address anywhere, and thus we have no way of reaching it from inside our program. This object is "lost" to us. (We will discuss what to do with "lost" objects shortly.)

If I were to ask you, "In a Java program, can I allocate an infinite number of objects?", you would instinctively know that the answer would be "no", because there is of course only a finite amount of memory. Now, you can see exactly what happens. With every call to `new`, another piece of dynamic memory is allocated, and so we keep moving "upward" in memory as we allocate more and more dynamic memory, until finally we run into the stack memory which was on the other side of the memory array. And, even if there were no stack memory being used we would eventually run into the other side – the "top" – of the memory array and we would *still* have no more memory left to allocate. So, just as runaway allocation from the stack (i.e. unstopped recursion) is bad, runaway allocation from the heap is bad as well.

Our earlier dereferencing works equally well here; the execution of the statement

```
(*intPtrOne) = 6;
```

begins with a dereference of `intPtrOne`, which will give us our recently allocated dynamic `int` "object" (at address **a10484** in Figures 2.31 - 2.34), since `intPtrOne` hold the address of that "object". Once that is done, you have a simple assigning of the value `6` to the dynamic "object", and the end result is that our new dynamic integer at address **a10484** now holds the value `6` (see Figure 2.35).

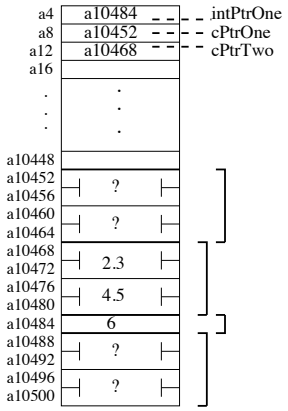

Figure 1.35: Our memory after the assignment of `6` to the dynamic integer

1.8 Garbage collection vs. `delete`

The last topic we need to deal with is the issue of "lost" objects, as seen in Figure 2.34. This is an important topic, and this is because we do not have an infinite amount of memory. Let's imagine that the 13 dynamically allocated cells pictured in Figure 2.34 are all the dynamic memory we have. (In real life, of course, you would have *way* more than 13 cells, but for the sake of ease-of-explanation, we'll keep our memory small for this hypothetical example. The ideas and concerns are exactly the same whether the memory is small or large.) In addition, imagine you wanted to dynamically allocate one more `Coord` object. Can you do it?

Well, as memory exists right now, you can't. You have four dynamic objects allocated – the `Coord` object at **a10488**, the `int` "object" at **a10484**, the `Coord` object at **a10468**, and the `Coord` object at **a10452**. All 13 of your dynamic cells are allocated.

However, recall that the first object we dynamically allocated, the `Coord` object at **a10488**, was "lost" to us. That is, even though it still existed in memory, we did not know its address and we had no way of referring to it from the rest of our program, and thus we had no way of using it any longer. By all rights, the four cells that this object takes up could be used for something else and our program wouldn't even notice that this object was gone. Such cells, rather than being called "lost", are generally referred to as *garbage*. So, our "lost" collection of cells can be called a *garbage object* or *garbage memory*.

Unfortunately, the cells are dynamically allocated already and it appears there is nothing we can do. Or perhaps there is. Let's ask the question, "What does it mean for a cell to be dynamically allocated?". Yes, it means we have set aside memory to store values, but we need to be a bit more specific. In particular, imagine that there is a "mark" on each cell in dynamic memory. This "mark" can have the value "in-use" or the value "not-in-use". When

your program first runs, all the cells are marked "not-in-use", and whenever you dynamically allocate a group of cells, part of what you are doing is changing the "marks" on those cells to "in-use". This "in-use" mark prevents the machine from giving a cell to a second object once it has already given that cell to an earlier object. Once a cell is part of an object, that cell cannot be given to any other objects until it is once again marked as being "not-in-use".

So, it would seem that the secret to reusing this garbage cell is to somehow mark it as "not-in-use". This should be okay to do; after all, it *isn't* in use anymore, so why not mark it accordingly? The problem is, how do we do this? How do we mark a cell "not-in-use"? How can we even tell that it isn't in use anymore? That's an important question, because we can't very well go setting these cell marks to indicate that the cell is no longer in use if, in fact, it still *is* in use. That could lead to the same problem we had when we returned addresses of local variables as we exited functions – namely, that we are trying to use memory that has been reclaimed by the system and is now possibly being used for something else. That is *always* bad.

So, what we need is some sure-fire means of marking as "not-in-use" only those cells that are indeed no longer in use. Then, the system can reclaim those cells (never mind how) and allocate them again when another request is made for more dynamic memory.

It turns out that there are two overall methods of doing this, and Java and C++ each use a different method. Recall that in Java, you may have used `new` quite a bit, and allocated many different objects, but when you were done with those objects, you just forgot about them. You moved on to other parts of your function, and forgot all about that lowly object that the reference `c1` referred to. Sure, maybe you set `c1` to refer to a different object, rather than just ignoring `c1` entirely, but that still left the object that `c1` originally referred to floating off in space. What happened to the object after that? Well, you didn't care. Your program didn't need it anymore, so you paid no attention to what happened to it. This is more or less what happened to our first object above. We didn't need it any more so we paid no attention to it. So, in Java, it appears that we could end up with certain memory cells being eternal garbage, because it doesn't appear as if we did anything about our garbage values.

Actually, the truth is more interesting. What happened to that garbage memory was that the Java interpreter eventually *collected* it, via a process known as *garbage collection*. A memory management system, such as the one inside an interpreter, has algorithms at its disposal that help it determine what memory is currently no longer being used and when it is okay to return those cells to the pile of "cells that are free for use".

A good-but-humorous analogy here is to think about a very young child. Imagine the child's parents are having company and have just cleaned the house. The child is young and oblivious to the world, and might well be gleefully runnning around the house, pulling toys off the shelf to play with and then quickly getting bored, tossing the toys over their shoulder, and running off in search of more toys. Meanwhile, the exasperated parents are chasing after the child, picking up the toys the kid left behind and putting them back where they belong. Now, this is not to say all Java programmers are childish :-), but the analogy is useful in that you can think of the Java interpreter as the parents. Your only goal in Java is to grab the memory and use it, and once you are done, you toss it aside and forget about it, and the Java interpreter runs around and collects it for you. "Collection" in this sense means marking the memory as "not-in-use", a marking that results in that memory being available for use by another dynamic allocation.

This method sounds reasonable enough. It also sounds pretty easy! Kick back, let the system collect your garbage for you...why would C++ want to do something different? Well, the problem with the earliest garbage collection algorithms was that they were *reliable*, but

inefficient. That is, though they made sure that you always had memory available to you to use for new data (if there was in fact any not-in-use memory somewhere in the system to begin with), there was work that needed to be done in order to collect some of your garbage memory. It took processor resources that you might otherwise have preferred to use for running your own programs. After all, the programmer knew exactly at what point the program was done with a particular piece of memory and no longer needed it. Any type of automatic garbage cleaning system is going to have to *detect* when a piece of memory is no longer needed, and however this is done, *some* programming resources – time, memory, or both – will need to be expended to acquire knowledge that the programmer already had.

So, went the argument back then, you *know* your program, you *know* the flow of logic, and you *know* when it would be appropriate to take an "in-use" dynamically allocated object and mark it as "not-in-use" once again. The system can't *know* these things, it can only *learn* them by means of various detection algorithms. And that learning process will take resources that you might be unwilling to give up.

So, you may prefer instead use your knowledge to handle the releasing of memory yourself, via direct instructions in the program to mark as "not-in-use" particular objects that were formerly "in-use". This is a process known as *freeing* memory or *releasing* memory or *deallocating* memory, or, in C++ slang, *deleting* memory . When you take responsibility for freeing dynamic memory, by deleting particular objects at particular times, the system will not need to consume resources to detect this information and collect unneeded cells, and thus your program is bound to be more efficient (or so the argument went back then...more on that in a bit).

However, this requires that you actually put those release instructions into the program, thus making your job harder because you now need to do work that before you could avoid entirely. In addition, you need to make sure you don't miss an object or two. If you forget to delete a dynamic object when you are done with it, then it just sits in the system, taking up memory. There is no halfway here, and no safety net – if you tell the garbage collector to go away, and attempt to handle releasing of memory yourself to save resources, then the garbage collector leaves for good (after all, what would be the point of running it in the background? It would defeat the purpose of doing it yourself) and you are responsible for deleting every last cell once you are done with it. If you don't, then you potentially end up with the runaway allocation problem, just as if you had no garbage collector in Java. And even if the eventual problem isn't quite that severe, your program will certainly consume more memory than it otherwise should, because it will need extra memory to hold all the extra objects that are garbage but are not being freed. When a program produces garbage data that it doesn't clean up, that program is said to *leak memory*, and the failure to release the memory is a problem often referred to as a *memory leak*.

So, that was the classic tradeoff. In some languages, dynamic memory was collected via garbage collection. This made the programmer's job easier, because it was no longer necessary to explicitly think about freeing dynamic memory. This meant the programmer could spend his or her time on implementing other parts of the design. In addition, it also meant that garbage was definitely collected – there weren't bits and pieces that were forgotten – because whatever faults the garbage collector might have had, its one big strength was that it was very thorough. Since the programmer can possibly make mistakes in this area, garbage collection was a way of avoiding the problems caused by forgetting to release a particular object. Memory leaks aren't going to happen in a system that uses a correctly-written garbage collection process.

But, on the other hand, garbage collection was rather slow at the time, and it did things its

own way and – depending on the algorithm – might use some memory overhead as well. If you didn't want that background collection process slowing your software down, and if you didn't want any of your memory wasted with collection overhead, then writing the explicit instructions to deallocate memory was the way to go. But, just as garbage collection had its disadvantages, explicit deallocation also had its disadvantages – namely, it was harder to take care of explicit deallocation than it was to just rely on the system, and also the process of explicit deallocation is inherently more error-prone than garbage collection because it is a lot easier for you to forget a deallocation than it is for the computer to run its algorithms incorrectly.

Since the early days, there has been a great deal of research into garbage collection systems, and more recent algorithms have been designed in a much more intelligent way, to the point where the overhead involved with garbage collection isn't always all that much worse than the overhead involved in just manipulating explicitly deallocated blocks. (That is, even manual memory management takes *some* time behind the scenes for record-keeping and a bit of memory manipulation; until recently, though, it was much less time than garbage collectors needed to do their garbage collection work.) However, there is nothing to keep you from building a garbage collection environment around a C++ program – that is, the freedom C++ gives you allows you to *choose* whether to handle memory manually everywhere, or whether to plug in a garbage-collection environment of your choice. You have access to the lower-level tools if you want them – though you don't always need them. So, though we will be dealing with explicit deallocation in this course, keep in mind that – in keeping with the "you make the choice" theme of C++ – you have the ability to build around your program whatever extra capabilities you might need, or to install libraries for your program to use that provide those extra capabilities for you. The more research into garbage collection that goes on, the more likely it is that choosing to compile such a system in with your program might be a good choice. Fortunately, you have that choice. Our focus here, though, will be on learning explicit deallocation, which is still a reasonable and/or superior choice in many circumstances and which is important for you to understand.

So, how do we handle this "explicit deallocation" in C++? It is done with the `delete` operation. Just as `new` was a function, `delete` is a function as well. The usage of `delete` is:

```
delete ptrToObject;
```

That is, on a line of code you have the word `delete` followed by a variable that holds an object's address – i.e. followed by a pointer variable. What delete will then do is:

1. (ignore this step) Call the object's destructor. Again, just as with `new` and constructors, we will get to this step later on. For now, you can ignore it – but I did want you to know it was there so it wouldn't be a surprise later on.

2. Free all the memory that was granted to this object when it was allocated using `new`.

So, if you have a pointer `objPtr` to a dynamic object, you would use the statement `delete objPtr;` to free the memory of that dynamic object. This action is often referred to with the phrase, "calling delete on the pointer `objPtr`", or just "calling delete on `objPtr`" or even simply "deleting the object". Regardless of the terminology used, the point is that you are marking that memory "not-in-use" so that the system can once again put it in the "pile of cells available for use". If all dynamic objects are deleted once their use is over, then we will never have any "lost" objects and therefore we will never "leak memory".

Note, however, that when we execute the statement `delete objPtr;`, *nothing happens to the pointer itself!!!*. This is *very* often a point of confusion with students, so take careful note of

it. When you call `delete` on a pointer, what you are doing is deleting the dynamic object whose address is held by that pointer. You are *not* eliminating the pointer variable itself. In fact, after the `delete` call, the pointer variable will *still* be holding the address of the now-deallocated dynamic memory. For example, if the seven line code snippet we used in the dynamic memory discussion (section 2.7) was changed to the following:

```
int* intPtrOne;
Coord *cPtrOne, *cPtrTwo;
cPtrOne = new Coord();
intPtrOne = new int();
cPtrTwo = new Coord();
cPtrTwo->Initialize(2.3, 4.5);
delete cPtrOne;
```

then immediately after the new line 7 (the `delete` call), the dynamic `Coord` object at **a10488** has been deallocated but the pointer `cPtrOne` still holds the address **10488**. In addition, `cPtrOne` can still be used as a variable in all the same ways that it could be used prior to the `delete` call. (See Figure 2.36.) Calling `delete` on a pointer variable does *nothing* to the pointer variable – it merely deletes the object whose address was held by that pointer variable.

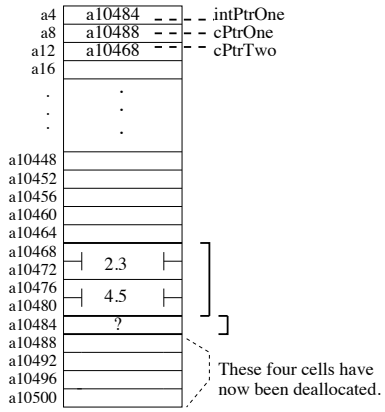

Figure 1.36: After the block of memory at **a10488** is deallocated via the statement `delete cPtrOne;`, `cPtrOne` itself still holds the address **a10488**.

If a pointer is storing the address of a now-deallocated object, as `cPtrOne` is in Figure 2.36, as a general rule it is important to assign some other address to this pointer variable, rather than allowing the pointer variable to continue to hold the address of garbage memory. If the pointer variable's scope is coming to an end – for example, if the pointer variable is a local variable of a function and you are about to return from that function – well, in that case the pointer variable is about to be deleted from the stack anyway, so you don't need to worry too much about what address it holds. But if the pointer variable is going to last for even some small time longer in your system, it is preferable to assign a new address to this pointer, so that you don't risk forgetting later on that the address in this pointer variable is meaningless. If you forgot that, you might end up dereferencing the pointer variable in an attempt to gain access to the object it refers to, but the object it refers to no longer exists! And again, accessing garbage memory is **BAD**.

What if we don't really *have* any other reasonable address we could store in this variable? For example, what if in the case of `cPtrOne` there were no more dynamically allocated `Coord` objects whose addresses could be assigned to `cPtrOne`? We can't just assign any random address to be stored there, because we'd have to remember that there wasn't really an object at that address, and therefore we'd still have the same problem. Fortunately, in C++, there is a solution to this problem. The value `NULL` in C++ represents a memory location that absolutely, positively, can *never* hold an object. Usually, `NULL` is equivalent to **a0**; that is, usually the very first cell in memory is set aside as the `NULL` cell and never serves any purpose other than to be a cell that stray pointers can point to in a standard manner. However, `NULL` *could* be some other cell, depending on what the system designers chose. The definition of NULL is implementation dependent.

This would appear to mean that setting a pointer to `NULL` is okay, whereas setting a pointer to **0** is not, because **0** is the bit pattern of all zeros, which is the address **a0**, and that address is not necessarily the "null address" on all implementations. However, it is indeed okay to assign the value **0** to a pointer if you want that pointer to hold the "null address". This is because the system interprets the symbol **0** to mean "null address" rather than "the bit pattern of all zeros" when dealing with pointers. Or, in other words, when assigning **0** to a pointer variable, the address that will get stored in that pointer variable is implementation dependent. It might be the address **a0**, or it might not be. It *will* be whatever address is the "null address" on that particular system.

So which is better, **0** or `NULL`? Or are they equivalent? Well, on most systems, they are exactly equivalent. This would suggest that using `NULL` is nicer, because you can tell at a glance you are dealing with pointers when you see `NULL`, whereas that is harder to do when you use **0**. (You will appreciate this benefit later.) However, on a few systems, `NULL` is defined in an odd way that can cause problems with some code. So, assigning the value **0** to pointer variables is always safe, whereas assigning the value `NULL` to pointer variables is only usually safe. We will use `NULL` in this class, so that you have the additional documentation provided by the use of a word rather than a number to aid you in understanding pointer code. However, keep in mind that to be truly portable, you either must use **0**, or else must redefine `NULL` in your code to be equal to **0**, thus overriding whatever oddity the system definition might have.

At the end of our code snippet above (the one with the `delete` call as line 7), we could have added the following line as line 8:

```
cPtrOne = NULL;
```

and this would assign the value `NULL` to `cPtrOne`, so that the pointer variable would not hold an address that was no longer the starting address of an allocated object. Note from Figure 2.37 that, since we can't be sure from system to system which address is considered to be `NULL`, our convention when drawing pictures of pointers that hold the value of `NULL` is not to write an address into the cell, but rather to draw a slash through the pointer variable itself. The slash indicates that this pointer has been set to `NULL`.

With the `NULL` value at our disposal, we have the tools we need to check for object existence in our program. Our programming discipline should be such that either a pointer holds an address of a real object, or else it holds `NULL`. This allows checks such as the following:

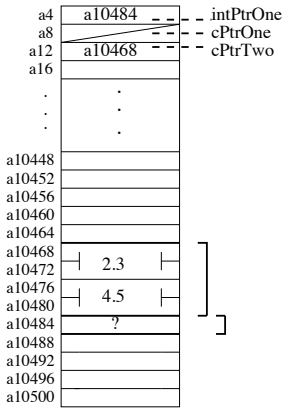

Figure 1.37: After the `delete` call, we can set `cPtrOne` to `NULL`. Note the slash through the cell at **a8**, which graphically indicates that this cell holds the `NULL` address.

```
if (objPtr != NULL)
   // do something with (*objPtr) here
else
   // do nothing, since (*objPtr) doesn't exist
```

As shown above, you can compare a pointer against `NULL` to determine if it holds the address of a real object or not, and take the appropriate action in each case. Of course, you may wonder how we know that any given non-`NULL` address is the address of a real object rather than the address of garbage information (as the pointer `cPtrOne` is in Figure 2.36). And the answer is...we don't. However, we would like to *assume* that any pointer that does not store the `NULL` address instead stores the address of a real object, and in order for that assumption to be correct, we would have to *always* avoid the case where a pointer holds the address of garbage data. The proper, disciplined use of `NULL` helps us avoid those kinds of errors, by allowing us to assign a standard, agreed upon "empty pointer" address to pointers which would otherwise store the addresses of garbage data.

1.9 A few final notes

1. When a pointer variable holds the memory address of an object, we often say that the pointer variable "points to" that object. For example, in section 2.5 we could have said that `numPtr` "pointed to" `n`, or, near the end of the section, that `cAddr` "pointed to" `c1`. Likewise, we could have said in the previous section that `cPtrOne` pointed to a dynamic object before the `delete` call was made. This is how we get the name "pointer". The conceptual picture is the same one you use when explaining the relationship in Java between references and the objects they refer to, and in fact we will often use exactly that conceptual picture in our C++ discussions, rather than drawing the memory diagrams we have been using up to this point. The memory diagrams are helpful when it comes to learning exactly what is going on, because if you at least have that knowledge in the back of your mind, many memory-related topics are easier to understand. But, when we sketch

out the pointer/object relationship, we often abandon the "real memory" diagrams and use an abstract picture such as the one in Figure 2.38.

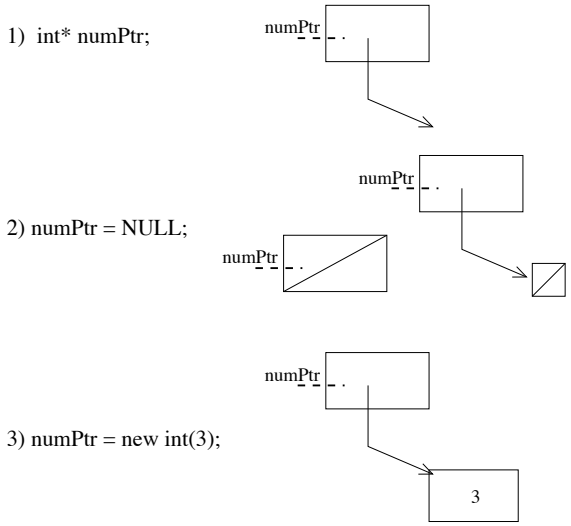

Figure 1.38: Our typical abstract picture. **(1)** The pointer is a box with an arrow. Right after allocation the pointer variable holds only garbage data – not a meaningful address – so the arrow in our picture points off into space. **(2)** When the pointer holds the value `NULL`, that is shown either by removing the arrow and putting a slash through the pointer box, as in the picture on the left, or else by leaving in the arrow but pointing it to a box with a slash through it, as in the picture on the right. **(3)** If a pointer holds the address of an object, that can be shown by having our arrow point to a second box, which represents the object. Inside this second box, we can store whatever values would be appropriate for the example.

2. Pointers are among the most frequent sources of errors in C++ programs. This is because, despite our best efforts, it is relatively easy to introduce defects into code which cause a pointer to hold an address where no object exists. The run-time errors known as **segmentation faults** and **bus errors** are related to the problem of trying to access an object via the address stored in a pointer when there is only garbage at that address. Be *very* careful when coding routines involving pointers.

3. Make sure you use addresses when appropriate and objects when appropriate. For example, if you have three integer pointers, `myIntPtr1`, `myIntPtr2`, and `myIntPtr3`, and each one points to a different integer, you cannot add the integers with the expression:

```
myIntPtr1 + myIntPtr2 + myIntPtr3
```

   The above expression will attempt to add the memory addresses. If you want to add the integer objects, you must de-reference the pointers to get those objects:

```
(*myIntPtr1) + (*myIntPtr2) + (*myIntPtr3)
```

This is a relatively simple example of this problem; you will encounter more complex examples throughout the semester. Basically, if the operations you are about to use are operations that work on objects instead of pointers, make sure to remember to dereference the pointers to get the objects, rather than incorrectly applying the operations to the pointers.

.