Lecture 27 — More About Pointers

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Acknowledgments: D.W. Harder, W.D. Bishop

Pointers & Memory Allocation

First, we examined memory-mapped I/O in which we had agreed upon an address in advance.

When we used a pointer related to a variable, we've just taken the address of an already existing variable (e.g., `int* ptr = &y;`)

This seems kind of redundant, because we could just use that variable's name, `y`, instead of using the pointer.

We often want to use pointers to allocate new variables at run-time.
Like an object or array, the keyword for that is `new`.

Run-Time Memory Allocation

```
int* ptr = new int( 256 );
```

This allocates memory for a new int and puts the value 256 in it.

Two separate memory locations have been allocated:

- The integer variable, containing 256; and
- The pointer `ptr`, containing the address of the integer.

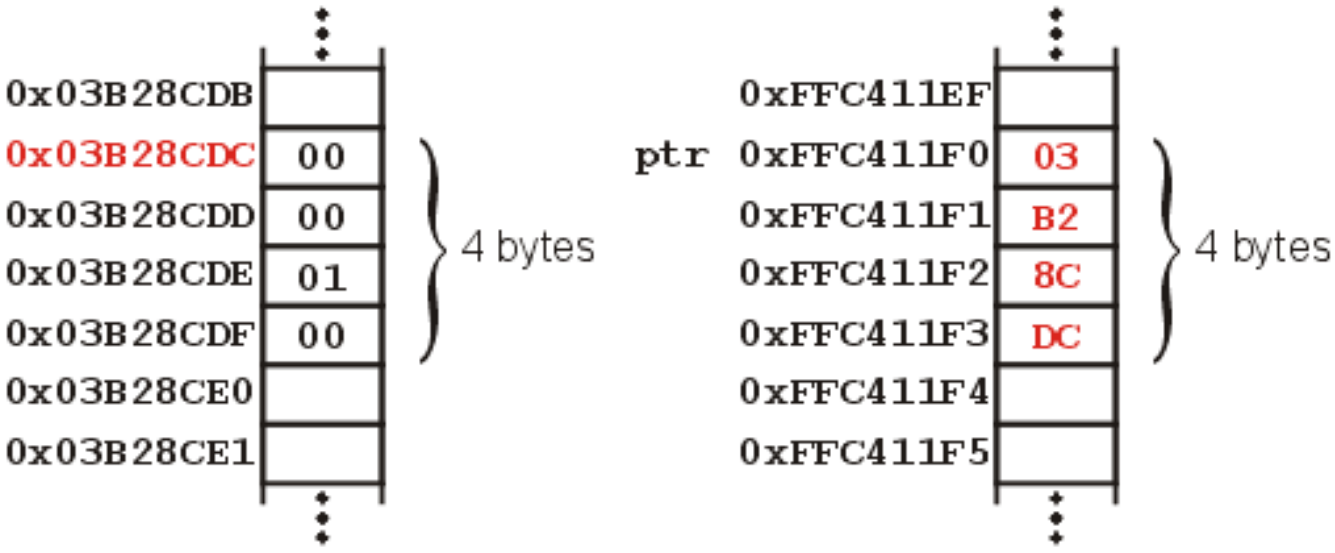

Image Source: D. W. Harder

Run-Time Memory Allocation

When the `new` keyword was used, the system searched for a block of free memory the size of an integer, and allocated it to our program.

The address of the memory allocated is then stored in `ptr`.

```
int* ptr = new int( 42 );
```

A simplified image to picture this statement is:

Image Source: D. W. Harder

Notice that unlike the usual situation when we allocate an `int`, there's no variable name associated with the integer variable.

Memory Leaks

We have already discussed the concept of memory leaks: an area of memory remains allocated even though it is no longer needed.

This happens if we allocate memory with `new` but forget to tell the system we are done with it when we are finished.

Here is some code that demonstrates how this might happen:

```
int* ptr = new int( 42 );
ptr = new int ( 1024 );
```

Memory Leaks

What happened to the memory allocated by the first call to `new`?

It's still there, still contains 42... but we have no way to access it.

We have lost the address of the variable that contains 42 and there's no way to find it again.

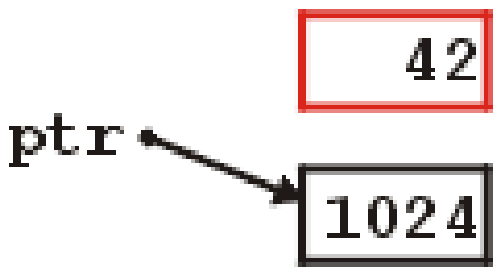

Image Source: D. W. Harder

Leaking an integer variable doesn't seem like a big deal given the size of computer memory, but leaks build up over time.

Memory De-Allocation

Even if we don't reassign `ptr` and lose the location of a variable, it's still necessary to free up memory when we're finished with it.

The keyword for freeing up that memory is `delete`.

Syntax: `delete ptr;`

Important: anywhere that `new` is used on a pointer requires a matching `delete`, otherwise we leak memory.

After use of `delete` on a pointer, remember to set that pointer to zero:

```
ptr = 0;
```

Image Source: D. W. Harder

Proper Use of Pointers

Why set pointers to 0 when `delete` has been called?

If you dereference a pointer containing the address 0, the operating system terminates the program with an error:

- In UNIX/Mac OS: “Segmentation Fault (Core Dumped)”;
- In Windows: You get the “report problem to Microsoft” dialog.

Thus, an attempt to dereference the pointer after it’s been deleted will result in an error and you will know immediately what went wrong.

Forgetting to Set the Pointer to 0

If you don't set the pointer to 0, it still contains the old address.

You have called `delete` so the memory is considered "free" and could potentially be allocated for some other purpose.

If you dereference this pointer, several things could happen:

- The memory is still there and you can still access it
- The memory may be re-allocated to another variable (so two pointers are pointing to the same thing, unexpectedly)
- The memory might be allocated to another program; an attempt to access it results in the OS terminating the program.

Forgetting to Set the Pointer to 0

Which of those three outcomes happens is totally random.

Murphy's Law says that the first one will happen in testing; one of the other two will take place when you release the software.

An error that happens only some of the time is much harder to identify and fix than a consistently-occurring error.

If the pointer is set to 0 after `delete` has been called, the outcome is consistent and we will be able to identify the error straight away.

Double `delete`

What if we did this?

```
int * ptr = new int( 42 );
delete ptr;
delete ptr;
```

This has the potential to very seriously confuse the system since we marked the memory as free twice.

The system keeps track of what memory is allocated and what is free;
Deleting the same location twice corrupts this bookkeeping data.

Double `delete`

What happens if we have corrupted the bookkeeping data about what memory is allocated and what is free?

The real answer is “undefined behaviour”:
We cannot predict what will happen.

If you are extremely fortunate it will have no impact.

If you are “lucky” the program will crash immediately and you will find the error quickly and correct it before any more damage is done.

More likely, you get negative consequences like silent data corruption.

The Power of Pointers

Because pointers give developers direct access to memory, they are very powerful tools.

Given a pointer to something on the program stack, it is possible to search for something nearby on the program stack.

The stack is, after all, just a designated area of memory.

Given an address in there, we could use pointers to look around.

Stack Hack Example

```
using System;

class StackHack
{
    unsafe static int EnterNumber( )
    {
        int result;

        Console.Write( "Enter a number: " );
               result = int.Parse( Console.ReadLine( ) );
        Console.WriteLine( "\nAddress of result on stack: " + (int) &result );
        HackIt( );

        return( result );
    }

    static void Main( )
    {
        int i = EnterNumber( );

        Console.WriteLine( "HackIt( ) already knew you entered " + i );
    }
```

Stack Hack Example

```
unsafe static void HackIt( )
{
    int x = 0;
    int* a = &x;

    Console.WriteLine( "Address of x on stack: " + (int)&x );

    for( int i = 0; i < 6; i++ )
    {
        Console.WriteLine( a[i] );
    }
    Console.WriteLine( "\nYou entered " + a[3] );
}
```

`HackIt( )` looks around the stack and knows where to find the number entered.

Comments on the Stack Hack Example

This example illustrates clearly why passwords stored as plaintext are vulnerable.

If several users were logged in at one time, it might be possible for one user to read confidential data of another.

References do not solve all security issues; the level of abstraction between references and pointers provides another layer of protection.

Pointers & Managed Data Structures

In C#, managed data structures (classes/objects) add another level of complexity when working with pointers.

How do you pass a pointer to a member field within a class?

If the data structure moves, the member field moves and the pointer will point to the wrong memory address!

Clearly, you either need to copy the member field to another variable or fix the location of the object in memory.

Without Fixing the Location

```
class Point
{
    public int x;
    public int y;
    public Point( int a, int b )
    {
        x = a;
        y = b;
    }
}
unsafe class FixedTest
{
    static void Main( )
    {
        Point myPoint = new Point( 2, 4 );
        int* ptr = &(myPoint.x);
        Console.WriteLine( "x = " + *ptr );
    }
}
```

Without Fixing the Location

This is a compile-time error in C#.

The compiler says we can only take the address of an unfixed expression inside of a fixed statement initializer.

Conclusion: we need to fix the location of the object in memory.

Pinning Managed Data Structures

C# allows developers to use a technique called pinning to fix the location of a reference type in memory.

The `fixed` keyword is used to temporarily pin a data structure in memory so that the data structure will not be relocated.

The `fixed` statement prevents relocation of a variable by the garbage collector.

The `fixed` statement takes the following form:

```
fixed( type* ptr = expr )
{
    // Statement block
}
```

Fixing the Location

```
class Point
{
    public int x;
    public int y;
    public Point( int a, int b )
    {
        x = a;
        y = b;
    }
}
unsafe class FixedTest
{
    static void Main( )
    {
        Point myPoint = new Point( 2, 4 );
        fixed( int* ptr = &(myPoint.x) )
        {
            Console.WriteLine( "x = " + *ptr );
            Console.WriteLine( "y = " + *(ptr + 1) );
        }
        Console.WriteLine( "x = " + *ptr );
    }
}
```

Pointer Arithmetic

In the example, we did something tricky called pointer arithmetic.

```
"y = " + *(ptr + 1)
```

Pointer arithmetic using arithmetic expressions involving pointers.

`(ptr + 1)` adjusts the address stored in the pointer.

Pointer arithmetic works based on the size of the pointer.

If an integer takes up 4 bytes in memory, then (ptr + 1) increments the address by 4 (because 4 is the size of an integer pointer).

When to Use Pointers

Obviously, if writing code in C or C++, there is no choice: use pointers.
But when should we use them in C#?

Generally speaking, use of pointers makes sense for:

- Legacy data structures
- Legacy library functions such as those provided by Windows
- Interoperating with other software (not C# programs you control)
- Performance critical code