Lecture 21 — Pointers

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 26, 2016

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

Introduction to Pointers

Pointers are considered one of the most difficult programming topics.
It is also one of the most important.

A pointer is, in simplest terms, the memory address of a variable.
It's called a pointer because it "points" to the variable's location.

Memory Addresses

Recall that when we use a variable or object, this data is stored somewhere in main memory.

When a variable `int x` is created, somewhere in memory, some area of memory is designated as the location of `x`.

This area of memory has an address. So to find `x` again when we need it, we go to the memory address of `x`.

Our model of memory is a linear array of "boxes".

- Each box is associated with a numerical address.
- Memory addresses are just numbers.

Pointer Variables

Even though a pointer is a memory address and a memory address is just a number, storing a pointer in a regular variable doesn't work.

Instead, declare a pointer type. This declares a pointer variable that points to a variable of type `int`:

```
int* p;
```

The use of the asterisk (*) indicates that `p` is a pointer.

Each variable type (`int`, `double`, `char`, etc...) requires a different pointer type.

Different Pointer Types

Why do the different variable types require different pointer types?

The size of a memory address is the same regardless of what kind of data you want to locate in memory.

We must tell the compiler what kind of pointer it is so the compiler knows what to do when we access whatever is stored at that address.

Pointer Declaration Pitfall

Although we don't do this in the lecture notes, it is legal to declare two `int` variables in one line, like this:

```
int x, y;
```

This is equivalent to:

```
int x;
int y;
```

When declaring a pointer you can write `int *x;`

- You may think of the "type" of a pointer as `int*`.
- This is dangerous: it's incorrect in C or C++.

Common source of confusion: writing `int* x, y;`

Pointer Declaration Pitfall

`int* x, y;` in C or C++ is equivalent to:

```
int *x;
int y;
```

`x` is an integer pointer, but `y` is an ordinary integer.

Pointer Declaration Pitfall

There could be confusion about how `int* x, y` will be compiled.

Code with the potential to cause confusion is not good practice; especially when working with others.

To avoid this problem altogether, don't put multiple variable declarations on the same line.

There is no extra cost for using another line in the source file.

I Know Your Address

Now, for a pointer to be useful, we need to put an address in it.

Perhaps we already know a specific address we would like to use, such as an address agreed upon in advance.

Example: an input/output device is mapped to a memory location.
This is known as memory-mapped I/O.

Memory-Mapped I/O

Memory-mapped I/O is a way of interacting with input/output devices.

A specific memory location is designated as belonging to a device.

Writing a value to the designated memory location of an output device results in the value being output via the device.

Reading a value from the designated memory location of an input device is how we receive input.

Writing to Memory-Mapped I/O

Let's say we have a parallel port mapped to memory location `0x378`.

In practice, this means to send output to the parallel port, we just write the value to location `0x378`, and that data gets sent to the port.

As we know the address that delivers the data to the parallel port, we can just set the pointer's address explicitly:

```
int* parallelPort;
parallelPort = 0x378;
```

This stores the memory address `0x378` in the pointer `parallelPort`.

Writing to Memory-Mapped I/O

Now we'd like to output the data 65 to the parallel port.

If `parallelPort` contains the address, we need a way to follow the pointer (the directions) to get to the memory location.

The syntax is: `*parallelPort = 65;`

This is called dereferencing a pointer.
Dereferencing is going to the address the pointer contains.

The use of the * in this context denotes dereferencing.
65 is written to the location pointed to by `parallelPort`.

Reading from Memory-Mapped I/O

Suppose the response to our action will be found in address `0x3A8`.

```
int* outputLocation = 0x3A8;
```

Allocates a pointer to an integer, and sets the address to `0x3A8`.

To read the value at `0x3A8`, once again, use the dereferencing operator *.

```
int latestValue = *outputLocation;
```

This dereferences the pointer `outputLocation`; the value there is, let's say, 128, so 128 is stored in the integer variable `latestValue`;

Summing Up Memory-Mapped I/O

The table below summarizes the expressions and values we have seen in the memory-mapped I/O example.

Expression	**Value**
`parallelPort`	`0x378` (Decimal: 888)
`*parallelPort`	`65`
`outputLocation`	`0x3A8` (Decimal: 936)
`*outputLocation`	`128`
`latestValue`	`128`

The variable `latestValue` is an `int`, not a pointer, and therefore it cannot be dereferenced.

What's Your Address?

Under most circumstances, we don't have an agreed-upon memory location that we know in advance.

There is a unary operator we can use to get the address of a variable.

The addressof operator is &

```
int y = 0;
int* ptr = &y;
```

The value stored in `ptr` is the address of `y`.

We now have two ways to refer to `y`:

Through the regular variable `y`; and

By following the address stored in `ptr`.

Address Of and Pointers

Now add a third statement to this list:

```
int y = 0;
int* ptr = &y;
*ptr = 42;
```

The value of `y` has been changed to 42 by the assignment. Why?

When we dereference `ptr`, we go to the memory location stored in `ptr`. The memory location in `ptr` is the address of `y`.

Pointer Assignment

If `p1` and `p2` are pointer variables, the assignment `p1 = p2` changes the address stored in `p1` to the address stored in `p2`.

Be sure not to confuse:

```
p1 = p2;
```

with

```
*p1 = *p2;
```

When you add the asterisk, you are not dealing with the pointers, but instead the variables to which they are pointing.

Pointers in C++/C

C and C++ use pointers to memory extensively.

A pointer provides the memory address of an a part of memory.

A pointer changes only when the program explicitly changes its value.

Pointer Code Example

```
void swap( int * a, int * b )
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main( )
{
    int x = 5;
    int y = 12;
    cout << "x = " << x << ", y = " << y << endl;

    swap( &x, &y );

    cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

Because the addresses of integers `x` and `y` were used, the values could be swapped without the use of pass-by-reference.