Lecture 22 — More About Pointers

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 28, 2016

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

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.
The keyword for that in C++ 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.

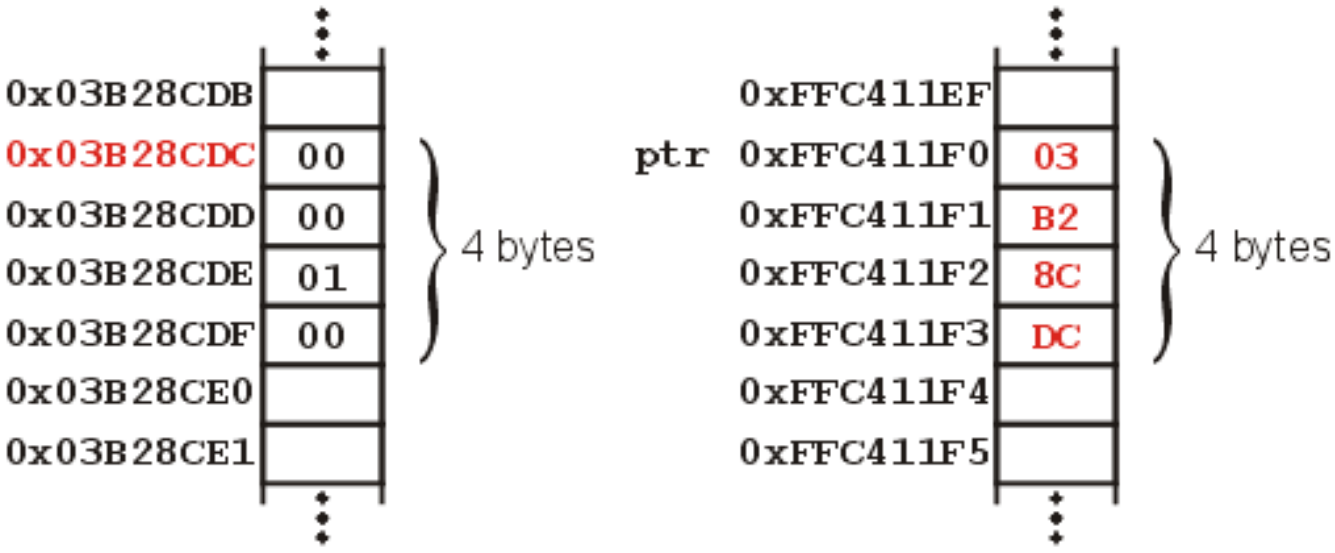

Image Source: D. W. Harder

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.

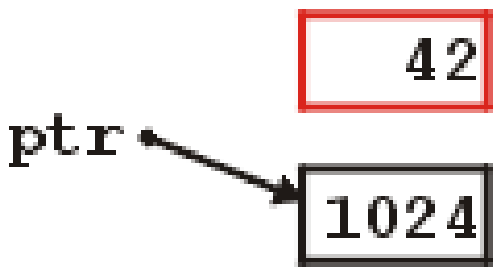

Image Source: D. W. Harder

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

```
int enter_number( )
{
    int result;
    cout << "Enter a number: ";
    cin >> result;
    cout << "Address of result on stack: << (int) &result << endl;
    hack_it();
    return( result );
}

int main( )
{
    int i = enter_number( );
    cout << "hack_it( ) already knew you entered << + i << endl;
    return 0;
}
```

Stack Hack Example

```
void hack_it( )
{
    int x = 0;
    int* a = &x;

    cout << "Address of x on stack: " << (int)&x << endl;
    for( int i = 0; i < 6; i++ )
    {
        cout << a[i] << endl;
    }
    cout << "You entered " << a[3] << endl;
}
```

`hack_it( )` 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.

Pointer Arithmetic

We can use pointers to allocate an array:

```
int * array = new int[10];
int* iterator = array;

for( int i = 0; i < 10; ++i) {
    *iterator = 0;
    iterator++;
}
```

Pointer Arithmetic

There is did something tricky called pointer arithmetic.
Syntax: `iterator++`

Pointer arithmetic is arithmetic expressions involving pointers.
`(ptr++)` 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 `iterator++` increments the address by 4 (because 4 is the size of an integer).

It allows us to walk through the array without use of the index operator `[]`.

Programmer Defined Types

As mentioned, we can allocate programmer-defined types (structs).

```
struct coordinates {
    int x;
    int y;
}
```

Consider the initialization: `coordinates* c1 = new coordinates;`

To set the coordinates:

```
(*c1).x = 5;
(*c1).y = 9;
```

This syntax prevents confusion between the precedence of the dereference operator (*) and the dot operator (`.`) in the statement.

The “cache miss operator”

This syntax is ugly, unfortunately.

But don’t worry – there is an alternative form: the arrow operator.

The arrow operator takes the form `->`.

It combines the dereference and the dot operator, in the right order.

To set the coordinates:

```
c1->x = 5;
c1->y = 9;
```

There’s no reason you must use the arrow operator; the alternative syntax is equally valid (but easier to make a mistake).

A Note About C

C++ is backwards compatible with the C programming language, so C syntax is valid in C++.

When allocating memory in C++, the `new` keyword figures out the correct amount of memory for our request. E.g. the size of an `int`.

In C the routine to allocate memory is a function `malloc( )`.

This function takes one parameter: the size of memory requested.

Use of `malloc`

If we know that an integer is 4 bytes, we may simply call `malloc()` with the parameter 4.

This is, however, bad practice. For compatibility and portability reasons, there is an additional operator to find the size of the type.

This is the `sizeof` operator, and it takes a type as its parameter.

```
int *x = malloc( sizeof( int ) );
```

Will allocate the correct amount of memory for an integer.

The `sizeof` operator can also be used on programmer defined types.

Use of `malloc`

For an array of capacity 10:

```
int *array = malloc( 10 * sizeof ( int ) );
```

So a memory array is just a pointer. And we use the capacity of the array as a multiplier times the size of the type.

Deallocation

Deallocation works similar to the `delete` keyword.

The function for deallocation is `free( )` and it takes a pointer.

Example: `free( array )`.

Just as with `new` and `delete`, every call to `malloc( )` must be matched to a call to `free( )`, otherwise memory is leaked!