ECE 150 The call stack and the heap

Douglas Harder

December 2023

1 Introduction

Every item of information must be stored as an integer, a floating-point number, a character or a Boolean value. Even if a piece of information is not actually an integer, but rather, you are only using the ones and zeros of an `unsigned int` to store other information, you must never-the-less use the types provided. These types must exist somewhere in main memory.

Not on the examination

When a program is executing, the computer uses *registers* to store information, and these registers are not in main memory, but rather are built into the processor itself. The compiler decides which values in memory are copied to registers, and when values in registers are copied back into main memory. Sometimes, the compiler can arrange it so that some local variables are unnecessary, and thus do not occupy main memory. Such optimizations, however, are way beyond the scope of a first-year course in programming. We will assume all local variables occupy memory.

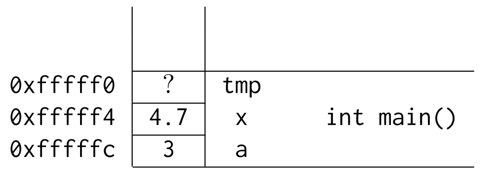
We will use the following class throughout these notes:

```
class Some_class {
    public:
        Some_class( int new_value );
        void zero();
        // Public member functions
    private:
        int value_;
        int square_;
        double data_[2];
};

Some_class::Some_class( int new_value ):
value_{ new_value },
square_{ value_*value_ },
data_{} {
    // Empty constructor
}
```

2 Local variables in `int main()`

If a local variable is declared in a function, the memory for that local variable is allocated *on the stack*. This begins with the function `int main()`. For example,

```
int main() {
    int a{ 3 };
    double x{ 4.7 };
    // some code...
    if ( ... ) {
        int tmp{ a };
    }
    // some more code...
}
```



Important

The order that the local variables appear on the stack is not relevant. You can put them in any order you want, only later, when we introduce parameters, parameters must come below local variables and local arrays. We are including addresses, for your interest, but they would not be required on any examination.

If `int main()` has a local array, then that, too, will be allocated on the stack; for example,

```
int main() {
    int a{ 3 };
    double a_integers[4]{ 1.2, 3.4, 5.6, 7.8 };
    // some code...
    if ( ... ) {
        int tmp{ a };
    }
    // some more code...
}
```

`0xffffd8`	?	tmp	
`0xffffdc`	1.2	a_integers	
`0xffffe4`	3.4		int main()
`0xffffec`	5.6		
`0xfffff4`	7.8		
`0xfffffc`	3	a	

If an object is declared as a local variable, sufficient memory on the stack would be allocated for all the member variables.

```
int main() {
    int a{ 3 };
    Some_class obj{ 5 };
    // some code...
    return 0;
}
```

`0xffffe4`	5	value_	obj	
`0xffffe8`	25	square_		
`0xffffec`	0.0	data_		int main()
`0xfffff4`	0.0			
`0xfffffc`	3		a	

In this example, the address of the object is `0xffffe4`, and the array `obj.data_` would be assigned `0xffffec`. Note that because `data_` is private, you can only access that array inside of a member function.

When a local variable that is an object (that is, declared to be an instance of a class):

- The constructor is called when the local variable is declared and the next statement does not execute until after the constructor returns. A call to the constructor is a function call.
- The destructor is called whenever a local variable that is declared goes out of scope.

If you declare an array of objects as a local variable, sufficient memory on the stack would be allocated for that number of instances of that class. For example,

```
int main() {
    Some_class a_objs[3]{4, 5, 6};
    // some code...
    return 0;
}
```

The constructor would, in this, be called three times, and each time the next entry in the array would be initialized:

0xffffb8	4	value_	a_objs	
0xffffbc	16	square_		
0xffffc0	0.0	data_		
0xffffc8	0.0			
0xffffd0	5	value_		
0xffffd4	25	square_		int main()
0xffffd8	0.0	data_		
0xffffe0	0.0			
0xffffe8	6	value_		
0xffffec	36	square_		
0xfffff0	0.0	data_		
0xfffff8	0.0			

Now, for example, `a_objs[2]` would access the object that has `value_` assigned 6.
When a local variable that is an array of objects:

- The constructor is called for each entry of the local array.
- The destructor is called on each entry of the array whenever a local array goes out of scope.

3 Function calls

In these examples, we will use the following three functions:

```
long abs( long const m ) {
    if ( m >= 0 ) {
        return  m;
    } else {
        return -m;
    }
}

long gcd( long m, long n ) {
    if ( m == 0 ) {
        return abs( n );
    }

    while ( n != 0 ) {
        long tmp{ n };
        n = m%n;
        m = tmp;
    }

    return abs( m );
}

void clear(
    double            array[],
    std::size_t const capacity
) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        array[k] = 0.0;
    }
}
```

When a function (that is, a function, constructor, member function or destructor) is called, memory for the parameters is allocated at the top of the stack, and those parameters are initialized with the arguments:

```
int main() {
    int var{ -25 };
    var *= abs( var + 19 );
    std::cout << var << std::endl;
    return 0;
}
```

0xfffff0	-6	m	long abs(...)
0xfffff8	-25	var	int main()

The parameter was initialized with the value of the argument, which is whatever returns when we calculate `var + 19`, which in this case is `-12`. When the function `long abs(...)` returns, the return value will be used to update the local variable `var`:

0xfffff8	-150	var	int main()

Next, if a function (that is, a function, constructor, member function, or destructor) has a local variable or a local array that is of a primitive data type or of a class, memory for those are allocated above the memory allocated for the parameters.

```
int main() {
    std::size_t const N{ 5 };
    double a_integers[N]{ 3.2, 4.7, 8.1 };
    clear( a_integers, N );

    return 0;
}
```

0xffffb8	?	k	
0xffffc0	0xffffd0	array	void clear(...)
0xffffc8	5	capacity	
0xffffd0	3.2	a_integers	
0xffffd8	4.7		
0xffffe0	8.1		int main()
0xffffe8	0.0		
0xfffff0	0.0		
0xfffff8	5	N	

When the loop begins executing, the loop variable `k` will be initialized to `0`. When `array[k]` is assigned to, it will be updating the entries of the array `a_integers`, so when the loop is about to exit, the call stack will look as follows, with the loop variable itself no longer accessible:

0xffffb8	5	k	
0xffffc0	0xffffd0	array	void clear(...)
0xffffc8	5	capacity	
0xffffd0	0.0	a_integers	
0xffffd8	0.0		
0xffffe0	0.0		int main()
0xffffe8	0.0		
0xfffff0	0.0		
0xfffff8	5	N	

When `void clear()` returns, we are back in the function `int main()`:

0xffffd0	0.0	a_integers	
0xffffd8	0.0		
0xffffe0	0.0		int main()
0xffffe8	0.0		
0xfffff0	0.0		
0xfffff8	5	N	

Let's look at a function calling another function, and where parameters have their values updated:

```
int main() {
    int first{ 204 };
    int second{ -150 };

    std::cout << gcd( first, second ) << std::endl;
    return 0;
}
```

0xffffd8	?	tmp	
0xffffe0	-150	n	long gcd(...)
0xffffe8	204	m	
0xfffff0	-150	second	int main()
0xfffff8	204	first	

At the end of the first iteration of the while loop, the parameters `m` and `n` have changed, but this has no effect in `int main()`:

0xffffd8	-150	tmp	
0xffffe0	54	n	long gcd(...)
0xffffe8	-150	m	
0xfffff0	-150	second	int main()
0xfffff8	204	first	

When the condition for the loop is false, the call stack now looks like the following, although once the loop finishes, the local variable `tmp` is no longer accessible.

0xffffd8	-6	tmp	
0xffffe0	0	n	long gcd(...)
0xffffe8	-6	m	
0xfffff0	-150	second	int main()
0xfffff8	204	first	

The next statement is "`return abs( m );`", so we call that function, and thus, we now have another layer on the stack:

0xffffd0	-6	m	long abs(...)
0xffffd8	-6	tmp	
0xffffe0	0	n	long gcd(...)
0xffffe8	-6	m	
0xfffff0	-150	second	int main()
0xfffff8	204	first	

The value `6` will be returned by `long abs(...)`, and this value is immediately returned by `long gcd(...)`, so this is what will be passed to the printing statement with `std::cout` inside of `int main()`.

4 Recursive function calls

Consider this recursive function:

```
int main();
int binomial( unsigned int n, unsigned int k );

int main() {
    // If your child has five toys, but is
    // only allowed to play with two of them
    // at a time, how many different ways
    // can you combine two toys from five?
    unsigned int toys{ 5 };
    unsigned int played_with{ 2 };

    std::cout << binomial( toys, played_with ) << std::endl;

    return 0;
}

int binomial( unsigned int n, unsigned int k ) {
    if ( k > n ) {
        return 0;
    } else if ( (k == n) || (k == 0) ) {
        return 1;
    } else {
        unsigned int first{ binomial( n - 1, k ) }
        unsigned int second{ binomial( n - 1, k - 1 ) };
        return first + second;
    }
}
```

With the first function call, the state of the stack is as follows:

0xffffe8	?	second	int binomial(5, 2)
0xffffec	?	first	
0xfffff0	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

The first two cascading conditions are false, so we are in the complementary alternative block, so must first perform the calculation for initializing `first`, and this has us call `binomial(4, 2)`:

0xffffd4	?	second	int binomial(4, 2)
0xffffd8	?	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

Once again, we are in the complementary alternative block, so now we must make another recursive function call to initialize the local variable `first` at `0xfffd8`:

0xffffc4	?	second	int binomial(3, 2)
0xffffc8	?	first	
0xffffcc	2	k	
0xffffd0	3	n	
0xffffd4	?	second	int binomial(4, 2)
0xffffd8	?	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

Again, we find ourselves in the complementary alternative block, so we must initialize the local variable `first` at `0xffffc8`, and this has us call `binomial(2, 2)`:

0xffffb4	?	second	int binomial(2, 2)
0xffffb8	?	first	
0xffffbc	2	k	
0xffffc0	2	n	
0xffffc4	?	second	int binomial(3, 2)
0xffffc8	?	first	
0xffffcc	2	k	
0xffffd0	3	n	
0xffffd4	?	second	int binomial(4, 2)
0xffffd8	?	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

At this point, the second condition is `true`, so we are in the second consequent block, which returns `1`. This value initializes `first` at address `0xffffc8`:

0xffffc4	?	second	int binomial(3, 2)
0xffffc8	1	first	
0xffffcc	2	k	
0xffffd0	3	n	
0xffffd4	?	second	int binomial(4, 2)
0xffffd8	?	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

The next statement in the current function call `binomial(3, 2)` is to initialize the local variable `second` at address `0xffffc4`, so this has us call `binomial(2, 1)`:

0xffffb4	?	second	int binomial(2, 1)
0xffffb8	?	first	
0xffffbc	1	k	
0xffffc0	2	n	
0xffffc4	?	second	int binomial(3, 2)
0xffffc8	1	first	
0xffffcc	2	k	
0xffffd0	3	n	
0xffffd4	?	second	int binomial(4, 2)
0xffffd8	?	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

Once again, we find ourselves in the complementary alternative block, so we must first initialize `first` at address `0xffffb8`, and we do this by calling `binomial(1, 1)`:

0xffffa4	?	second	int binomial(1, 1)
0xffffa8	?	first	
0xffffac	1	k	
0xffffb0	1	n	
0xffffb4	?	second	int binomial(2, 1)
0xffffb8	?	first	
0xffffbc	1	k	
0xffffc0	2	n	
0xffffc4	?	second	int binomial(3, 2)
0xffffc8	1	first	
0xffffcc	2	k	
0xffffd0	3	n	
0xffffd4	?	second	int binomial(4, 2)
0xffffd8	?	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

Fortunately, we are in second consequent block, as the parameters are equal, so the value `1` is returned, and that initializes the local variable `first` at `0xfffb8`:

0xffffb4	?	second	int binomial(2, 1)
0xffffb8	1	first	
0xffffbc	1	k	
0xffffc0	2	n	
0xffffc4	?	second	int binomial(3, 2)
0xffffc8	1	first	
0xffffcc	2	k	
0xffffd0	3	n	
0xffffd4	?	second	int binomial(4, 2)
0xffffd8	?	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

We must now initialize `second` at `0xffffb4`, so we call `binomial(1, 0)`:

0xffffa4	?	second	int binomial(1, 0)
0xffffa8	?	first	
0xffffac	0	k	
0xffffb0	1	n	
0xffffb4	?	second	int binomial(2, 1)
0xffffb8	1	first	
0xffffbc	1	k	
0xffffc0	2	n	
0xffffc4	?	second	int binomial(3, 2)
0xffffc8	1	first	
0xffffcc	2	k	
0xffffd0	3	n	
0xffffd4	?	second	int binomial(4, 2)
0xffffd8	?	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

Fortunately, we are in the second consequent block, and thus again we return `1`, and this initializes `second` at `0xffffb4`:

`0xffffb4`	1	`second`	`int binomial(2, 1)`
`0xffffb8`	1	`first`	
`0xffffbc`	1	`k`	
`0xffffc0`	2	`n`	
`0xffffc4`	?	`second`	`int binomial(3, 2)`
`0xffffc8`	1	`first`	
`0xffffcc`	2	`k`	
`0xffffd0`	3	`n`	
`0xffffd4`	?	`second`	`int binomial(4, 2)`
`0xffffd8`	?	`first`	
`0xffffdc`	2	`k`	
`0xffffe0`	4	`n`	
`0xffffe4`	?	`second`	`int binomial(5, 2)`
`0xffffe8`	?	`first`	
`0xffffec`	2	`k`	
`0xfffff4`	5	`n`	
`0xfffff8`	2	`played_with`	`int main()`
`0xfffffc`	5	`toys`	

Both local variables have been initialized, so now we can return their sum, and that sum initializes the local variable `second` at `0xffffc4`:

`0xffffc4`	2	`second`	`int binomial(3, 2)`
`0xffffc8`	1	`first`	
`0xffffcc`	2	`k`	
`0xffffd0`	3	`n`	
`0xffffd4`	?	`second`	`int binomial(4, 2)`
`0xffffd8`	?	`first`	
`0xffffdc`	2	`k`	
`0xffffe0`	4	`n`	
`0xffffe4`	?	`second`	`int binomial(5, 2)`
`0xffffe8`	?	`first`	
`0xffffec`	2	`k`	
`0xfffff4`	5	`n`	
`0xfffff8`	2	`played_with`	`int main()`
`0xfffffc`	5	`toys`	

For the function call `binomial(3, 2)`, we also have now initialized both local variables, so once again, we can return their sum (`3`), and this can now initialize the local variable `first` at `0xffffd8`:

0xffffd4	?	second	int binomial(4, 2)
0xffffd8	3	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

At this point, we must now initialize the second local variable `second` at `0xffffd4`, and to do this, we must call `binomial(3, 1)`:

0xffffc4	?	second	int binomial(3, 1)
0xffffc8	?	first	
0xffffcc	1	k	
0xffffd0	3	n	
0xffffd4	?	second	int binomial(4, 2)
0xffffd8	3	first	
0xffffdc	2	k	
0xffffe0	4	n	
0xffffe4	?	second	int binomial(5, 2)
0xffffe8	?	first	
0xffffec	2	k	
0xfffff4	5	n	
0xfffff8	2	played_with	int main()
0xfffffc	5	toys	

And you can take it from here!

5 Memory allocated on the heap

Each time you call `new typename{...}` or `new typename[N]{...}`, this makes a request to the operating system for sufficient memory for what was requested. The location of this memory is chosen by the operating system, and it is that address that is returned by the call to `new`.

This is important: both allocating a single instance of a type or allocating an array of a type returns an address. To help you remember which, you should prefix the pointer with a `p_` and prefix the latter with `a_`. For example,

```
int main() {
    int *p_integer{ new int{ 42 } };
```

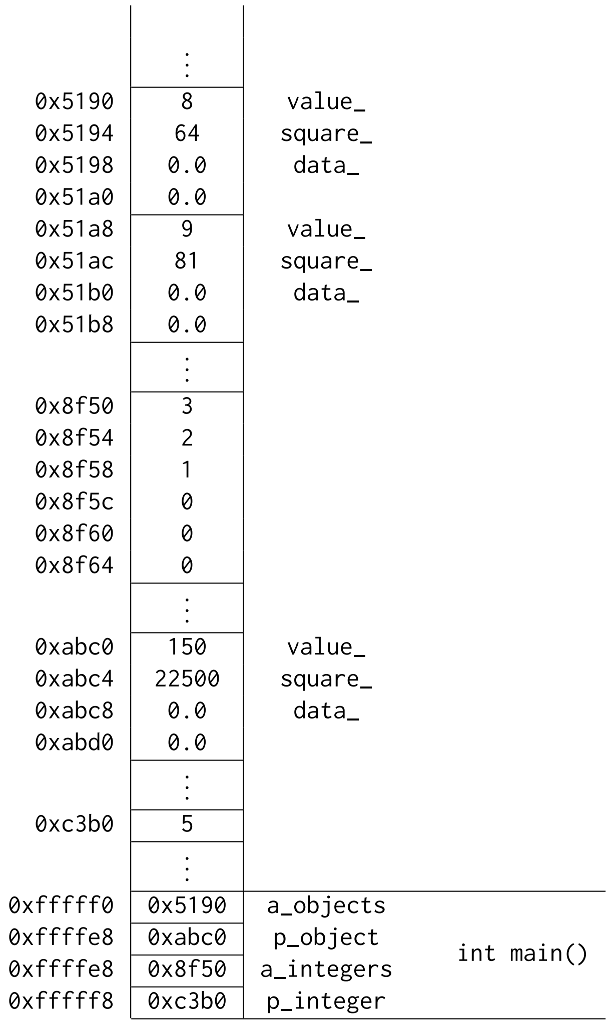
```
        int *a_integers{ new int[6]{ 3, 2, 1 } };
        int *p_object{ new Some_class{ 150 } };
        int *a_objects{ new Some_class[2]{ 8, 9 } };
        // Some code...
        delete p_integer;
        p_integer = nullptr;
        delete[] a_integers;
        a_integers = nullptr;
        delete p_object;
        p_object = nullptr;
        delete[] a_objects;
        a_objects = nullptr;
        return 0;
    }
```

The four pointers are actually local variables occupying however much memory is needed, and we will assume this is eight bytes (64 bits).

Not on the examination

On a 64-bit processor, both pointers and `std::size_t` occupy eight bytes (64 bits). On a 32-bit processor, both pointers and `std::size_t` occupy four bytes (32 bits). Many microcontrollers may have 32-bit addresses, 24-bit addresses or even 16-bit addresses, in which case, the memory allocated for a pointer and `std::size_t` would match that value.

Thus, we have the following situation:



You don't care what the addresses are. You only need the addresses so that you can access what is at that location.
For the address of a single instance of a primitive data type, you proceed as follows to access or assign to the instance at that address:

```
// Access or assign to an instance
// of a primitive data type
std::cout << *p_integer << std::endl;
*p_integer = 91;
```

The first will print what is at the address stored in the local variable `p_integer`, while the second changes what is at that address.
For an array of a primitive data type, you access or manipulate the array entries using array indexing:

```
    // Access or assign to an entries
    // of a dynamically allocated array
    // of a primitive data type
    std::cout << a_integers[0]
    for ( std::size_t k{ 1 }; k < 6; ++k ) {
        std::cout << ", " << a_integers[k];
    }

    for ( std::size_t k{ 0 }; k < 6; ++k ) {
        a_integers[k] = 1000 + k;
    }
```

The first loop prints `3, 2, 1, 0, 0, 0` while the second changes the values stored at addresses `0x8f50` through `0x8f64` to the values `1000` through `1005`, respectively.
To call a member function on the object at the address of a single instance of that class, you would proceed as follows:

```
    (*p_object).member_function(...);
```

This is awkward, so an easier method is to use the arrow operator:

```
    p_object->member_function(...);
```

To call member function on an array of dynamically allocated instances of a class (that is, an array of objects), remember that `a_objects[2]` is the object at the third entry. Thus, you must use the dot operator to call the member function on that object:

```
    for ( std::size_t k{ 0 }; k < 2; ++k ) {
        a_objects[k].member_function(...);
    }
```

Not on the examination

You may note that arrays and pointers are the same: they are addresses. Thus, you may ask, why not treat a single instance as an array of capacity 1, and then access it using the indexing operator: for example, `p_integer[0]` instead of `*p_integer`, and `p_object[0].member_function(...)` instead of `p_object->member_function(...)`?
Similarly, you may ask, can you not access the first entry in an array using the notation for pointers? For example, using `*a_integers` instead of `a_integers[0]`, and `a_objects->member_function(...)` instead of `a_objects[0].member_function(...)`?
The answer is, you can, but you will annoy everyone around you. If a pointer is being de-referenced using an index, the assumption is that that address is that of an array. The reader will then search in vain for the capacity of that array. Similarly, if you just want to access the first entry of an array by using `*a_integers`, then the reader will be even more confused if you are accessing other entries using array indices.

6 Calling delete or delete[]

When you call `delete` on a single instance of a class that was dynamically allocated, then that will call the destructor on that one instance. If you call `delete[]` on an array that was dynamically allocated, then that will call the destructor on each instance in the array. If you accidentally call, for example, `delete a_objects;`, this will only call the destructor on the first entry of the array, possibly leading to memory leaks.

Not on the examination

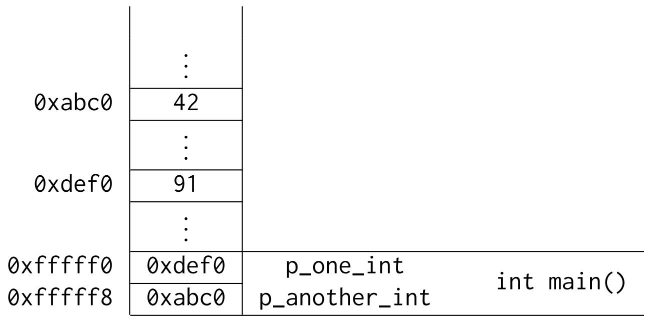
You may have noticed that the purpose of calling `delete[] a_objects;` is to call the destructor on all the entries of the array. Because there is no destructor for primitive data types, can you not just delete a simple array with `delete`? For example, `delete a_integers;`? Similarly, if the class has no destructor, could you not call `delete a_objects;`? The answer is, "yes", but then you run the risk of confusing the reader and possibly forgetting whether or not this is an array of objects or an array of primitive data types, and similarly, suppose later you add a destructor to a class: do you now have to go back and update all calls to `delete`?
It is always best practice to call `delete` on single instances and `delete[]` on arrays. In this class, we will generally not mark code wrong if it actually does compile and is valid; however, in the case of calling either `delete` or `delete[]`, we will be making sure that the expected call is made (the first on instances, the second on arrays).

7 When to call delete?

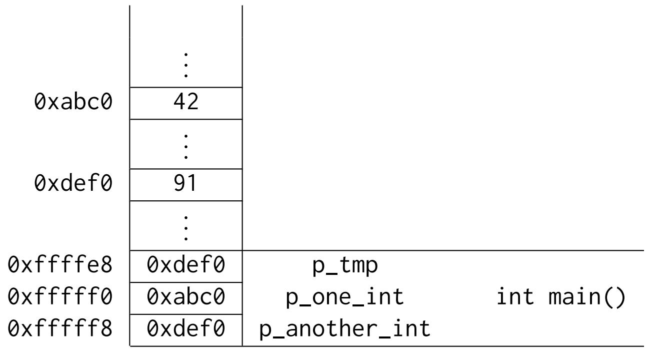
Suppose you have two pointers that are storing the addresses of two dynamically allocated objects and you want to swap them.

```
int main() {
    int *p_one_int{ new int{ 91 } };
    int *p_another_int{ new int{ 42 } };

    if ( *p_one_int >= *p_another_int ) {
        int *p_tmp{ p_one_int };
        p_one_int = p_another_int;
        p_another_int = p_tmp;
        // Do you call delete p_tmp; ?
    }

    delete p_one_int;
    p_one_int = nullptr;
    delete p_another_int;
    p_another_int = nullptr;

    return 0;
}
```

Consider what is happening in main memory:



With the conditional statement being executed, a local variable `tmp` is created and assigned, and at the end of the conditional statement, the stack looks like the following:



You will see that both local variables `p_tmp` and `p_another_int` now store the same address. If you now call `delete p_tmp;`, this will deallocate the memory that is also stored in the local variable `p_another_int`, in other words, `p_another_int` unexpectedly became a dangling pointer. Finding such a bug could actually be a serious problem, because later, when the source actually calls `delete p_another_int;`, this will cause the operating system to end your program, throwing an error.

Note that the above code does not swap the values stored, it swaps the two addresses. If you wanted to swap the values stored at those addresses, you would use:

```
if ( *p_one_int >= *p_another_int ) {
    int tmp{ *p_one_int };
    *p_one_int = *p_another_int;
    *p_another_int = tmp;
}
```