Highlights

- pointers
  ```c++
  int x = 6;
  int* xp;
  xp = &x;
  ```

- dynamic arrays
  ```c++
  int* x = new int[5];
  x[0] = 2;
  x[1] = 7;
  // ...
  delete [] x;
  ```

- new & delete
  ```c++
  int *xp;
  xp = new int;
  *xp = 5;
  delete xp;
  ```
An object is simply a box in memory and if you pass this into a function it makes a **copy**.

A memory address is **where a box is located** and if you pass this into a function, you can change the variable everywhere.

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Object (box)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrays</td>
<td>int, double, char, ...</td>
</tr>
<tr>
<td>using &amp;</td>
<td>classes</td>
</tr>
<tr>
<td>(pointers)</td>
<td></td>
</tr>
</tbody>
</table>
Review: address vs value

Consider the following:

```cpp
int x=6;
cout << x << "\n";
cout << &x << endl;
```

x is a variable (a box containing value 6)

&x is a memory address (sign pointing to box)
  - Rather than giving the value inside the box, this gives the whole box
(see: memAddress.cpp)
Review: address vs value

Similar to a URL and a webpage
- A URL is not a webpage, but a link to one

Webpage g;
cout << &g;
Pointers

Just as `&` goes from value (webpage) to address (url), `*` goes the opposite:

Webpage \( g; \)
URL \( u = \&g; \)
Webpage \( g2 = *u; \)
Pointers

You can also think of pointers as “phone numbers” and what they point to as “people”

1-800-presdnt (pointer)

Trump (object)
Pointers

If multiple people have the same “phone number”, they call the same person (object)

1-800-presdnt (pointer/memory address)

Trump (object)

1-800-presdnt
A **pointer** is used to store a memory address and denoted by a * (star!)

```cpp
int x = 6;
int* xp;
xp = &x;
```

Here variable “xp” has type “integer pointer”

```cpp
cout << *(&x); // *(&x) same as x
```

The * goes from address to variable (e.g. like hitting ENTER on a url, or “call” on a phone contact)  
(See: pointerBasics.cpp)
Pointers (phone analogy)

Make a phone-number for an person (int)

```
int* jacky;
```

Make a contact name called “jacky”

```
int Jackeline_Wu = 9;
```

Make a person (int) “Jacqueline Wu” exist

```
jacky = & Jackeline_Wu;
```

(& = address of)

Save Jacqueline Wu's phone number into the “jacky” contact

```
*jacky = 9001;
```

(* = call up)

Call the “jacky” contact (and connect with Jacqueline Wu)
Pointers

It is useful to think of pointers as types:

```c
int* xp;
```

Here I declared a variable “xp” of type “int*”

Just like arrays and [], the use of the * is different for the declaration than elsewhere:

Declaration: the * is part of the type (`int* xp;`)

Everywhere else: * follows the pointer/address (i.e. `*xp = 2;` puts 2 where xp is pointing to)
Pointers and references allow you to change anything into a memory address that you want. This can make it easier to share variables across functions.

You can also return a pointer from a function (return links to variables) (see: returnPointer.cpp)
Pointers

Why do we need pointers? (memory addresses are stupid!!)

Suppose we had the following class:

class Person{
    string name;
    Person mother;
    Person father;
};

Will this work?
Pointers

As is, it will not... it is impossible to make a box enclose two other equal sized boxes.

The only way it can enclose something like itself is that thing is smaller.
To do this we can use pointers instead!

A pointer does not store the whole class data, it only remembers where it is (like a URL)

```cpp
class person{
    string name;
    person* mother;
    person* father;
};
```

(See: person.cpp) (more on this shortly)
When dealing with classes, often you need to deference (*) and access a member (.)

There is a shortcut to de-reference and call a member (follow arrow and go inside a box)

You can replace (*var).x with var->x, so...

```cpp
(*me.mother).name;
```

... same as ...

```cpp
me.mother->name;
```
Person class

How would you make your grandmother? How could you get your grandmother using only yourself as a named object?

```cpp
class person{
    string name;
    person* mother;
    person* father;
};
```

(See: personV2.cpp)
What is comes next in this pattern?

Basic programming: `int x;`
Ask for one box with a name

Intermediate programming: `int x[20];`
Ask for multiple boxes with one name

Advanced programming: ???
???
Boxes

What is coming next in this pattern?

Basic programming: \( \text{int } x; \)
Ask for one box with a name

Intermediate programming: \( \text{int } x[20]; \)
Ask for multiple boxes with one name

Advanced programming: \( \text{new int; } \)
Ask for a box without giving it a name
Pointers are also especially useful to use with the `new` command.

The new command will create a variable (box) of the type you want:

```cpp
int x;
x = 2;
```

The new integer has no separate name, just part of `xp` (as array boxes part of array name) (See: `newMemory.cpp`)
What does this do?

```c
int main()
{
    while (true)
    {
        int *x = new int;
    }
    return 0;  // totally going to get here!
}
```
What does this do?

```cpp
int main()
{
    while(true)
    {
        int *x = new int;
    }
    return 0;  //totally going to get here!
}
```

Asking for a lot of boxes there...
(See: memoryLeak.cpp)
When your program exits, the operating system will clean up your memory.

If you want to clean up your memory while the program is running, use `delete` command.

```cpp
int *imaPointer; // pointer box (holds address)
imaPointer = new int; // point here!
// do some stuff...
delete imaPointer; // goodbye pointer
```

(See: deleteMemory.cpp)
delete

This is also a memory leak:

```cpp
int *ptr; // make a pointer
ptr = new int; // point here
ptr = new int; // more the merrier
delete ptr; // ERASE
```

By the 3\textsuperscript{rd} line, there is no link back to the box on the 2\textsuperscript{nd} line (dangling pointer)

There should be a “delete” for every “new”
As you can manage how you want to create new variables/boxes, using new/delete is called **dynamic memory**.

Before, the computer took care of memory by creating variables/boxes when you use a type then deleting when the function ends.
Memory management is a hard part of C++

You need to ensure you delete all your boxes after you are done with them, but before the pointer falls out of scope (see: lostPointer.cpp)

Some other languages manage memory for you.
The ability to have non-named boxes allows you to more easily initialize pointers

```cpp
class person{
    string name;
    person* mother;
    person* father;
};
```

(See: personV3.cpp)
You can have multiple stars next to types:

\[ \text{int}^{***} \ x; \]

Each star indicates how many arrows you need to follow before you find the variable

\[ \text{int}^{***} \quad \text{int}^{**} \quad \text{int}^* \quad \text{int} \]

\( x \)

(See: pointerPointers.cpp)
## What pointers can/cannot do

<table>
<thead>
<tr>
<th>Pointers CAN do</th>
<th>Pointers CANNOT do</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int</strong> <em>ptr</em>;</td>
<td><strong>int</strong> <em>ptr</em>;</td>
</tr>
<tr>
<td><strong>int</strong> x = 2;</td>
<td><em>ptr</em> = new <strong>int</strong>;</td>
</tr>
<tr>
<td><em>ptr = &amp;x;</em></td>
<td><em>ptr=3;</em></td>
</tr>
<tr>
<td>// pointer to...</td>
<td><strong>int</strong> x;</td>
</tr>
<tr>
<td><strong>int</strong> <strong>ptr2</strong>;</td>
<td><strong>int</strong> x;</td>
</tr>
<tr>
<td>// . . . a pointer!</td>
<td><em>ptr</em> = &amp;x;</td>
</tr>
<tr>
<td>int x = 10;</td>
<td>// cannot relabel/move box</td>
</tr>
<tr>
<td>ptr2 = &amp;ptr;</td>
<td></td>
</tr>
<tr>
<td>ptr = &amp;x;</td>
<td></td>
</tr>
<tr>
<td>// may seem weird...</td>
<td></td>
</tr>
</tbody>
</table>
When you type this, what is ptr pointing at?

```c
int *ptr;
```

Answer: nullptr (or NULL)

```c
int *ptr = nullptr;
```
nullptr

The null pointer is useful to indicate that you are not yet pointing at anything.

However, if you try to de-reference it (use *), you will seg fault.

```cpp
int *ptr = nullptr;
cout << *ptr << endl;
```

Do not try to ask the computer to go here.

(see: nullptr.cpp)
Multiple deletes

Every new should have one corresponding delete command (one for one always)

The delete command gives the memory where a variable is pointing back to the computer

However, the computer will get angry if you try to give it places you do not own (i.e. twice)

```c
int* x = new int;
delete x;
delete x;
```
Dynamic arrays

Arrays are memory addresses (if you pass them into function you can modify original)

So we can actually make a dynamic array in a very similar fashion

```cpp
int x;
cin >> x;
int *list; // pointer to array
list = new int[x];
// arrays are just memory addresses
```

(this memory spot better to store large stuff)
Dynamic arrays

One important difference to normal pointers

When you delete an array you must do:

```cpp
int *list; // pointer to array
list = new int[x];
delete [] list; // need empty square brackets
```

If you do the normal one, you will only delete a single index (list[0]) and not the whole thing

```cpp
int *list; // pointer to array
list = new int[x];
delete list; // BAD BAD BAD BAD BAD BAD
```

(See: dynamicArrays.cpp)
Another issue with arrays is that we could not return them from functions.

Since arrays are memory addresses, we would only return a pointer to a local array.

However, before this local array would just fall out of scope, but no more as dynamic memory stays until you manually delete it. (See: returnArrays.cpp)
Dynamic 2D arrays

Since pointers can act like arrays... (i.e. int* acts like int [])

... int** can act like a two dimensional array

But need to use new to create each column individually (but can change the size of them)

When deleting, same structure but backwards (delete each column, then rows)
Dynamic 2D arrays

```c++
int** arr;
arr = new int*[4]; // 4 rows (of pointers)
arr[0] = new int[4]; // 1st row = 4 cols
arr[1] = new int[5]; // 2nd row = 5 cols
arr[2] = new int[4]; // 3rd row = 4 cols
arr[3] = new int[3]; // 4th row = 3 cols
```

(See: raggedArray.cpp)
Dynamic 2D arrays

```cpp
int** arr;
arr = new int*[4]; // 4 rows (of pointers)
arr[0] = new int[4]; // 1st row = 4 cols
arr[1] = new int[5]; // 2nd row = 5 cols
arr[2] = new int[4]; // 3rd row = 4 cols
arr[3] = new int[3]; // 4th row = 3 cols
```

(See: raggedArray.cpp)
Reasons why pointer

Why use pointers?

1. Want to share variables (multiple names for the same box)
2. Dynamic sized arrays
3. Return arrays from functions (or any case of keep variable after scope ends) (DOWN WITH GLOBAL VARIABLES)
4. Store classes within themselves
5. Automatically initialize the number 4 above