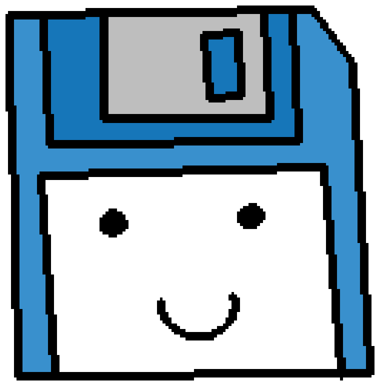
Pointers

Written by Rachel J. Morris, last updated 2016-02-14

Memory Addresses



Memory Addresses

When we declare a variable, a set amount of memory is set aside to store the value of that variable.

Different data types need different amounts of memory.

Characters and Booleans take 1 byte (8 bits),
Integers and floats take 4 bytes,
And Doubles take 8 bytes.

4 bytes

Memory Addresses

When we set aside memory for a variable, that means we also reserve a **memory address** for that item.

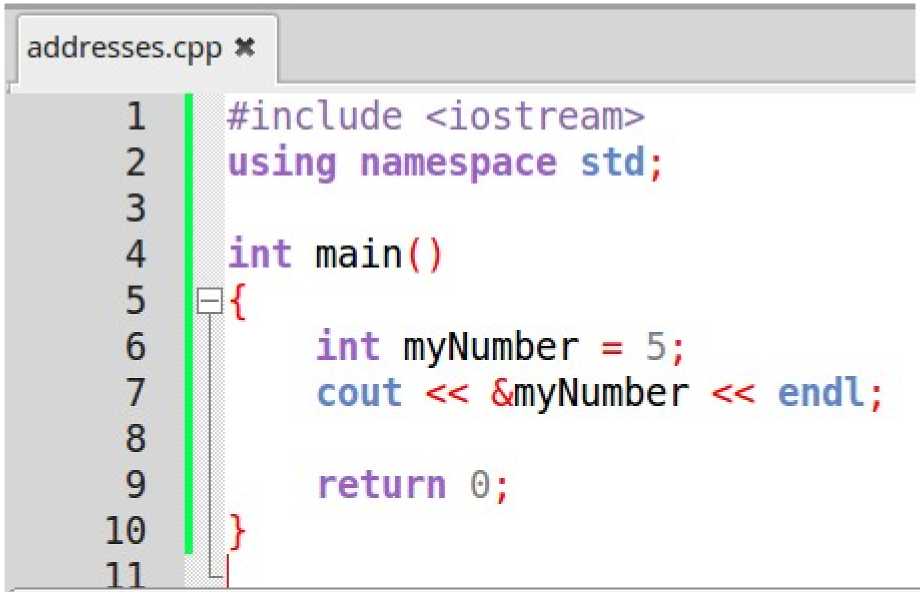
A memory address looks like a hexadecimal number.

Address:
`0x7fff957449bc`

4 bytes

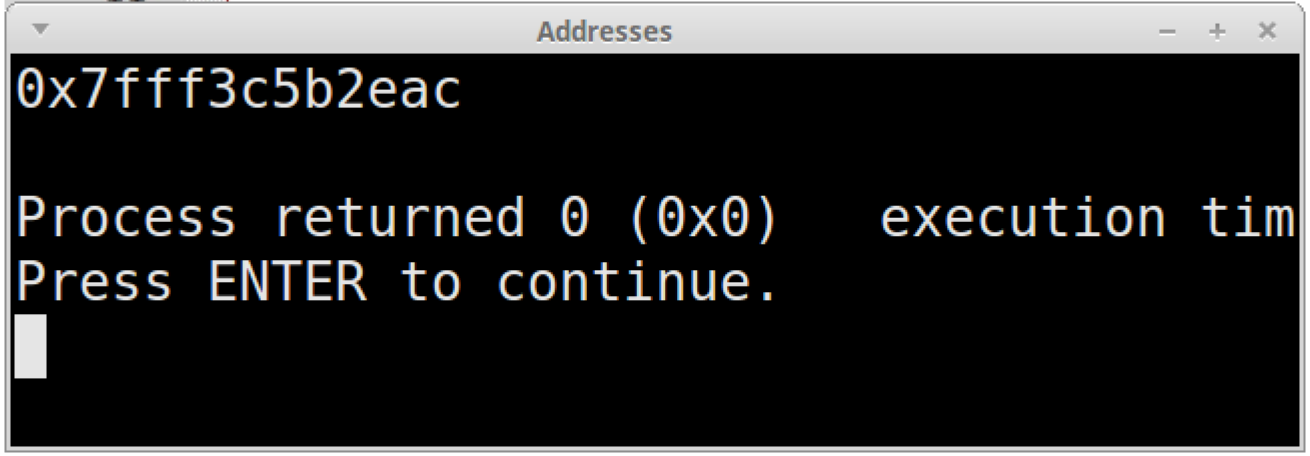
Memory Addresses

In C++, you can view a variable's memory address by using the address-of operator & , which prepends the variable's name.



addresses.cpp

```cpp
#include <iostream>
using namespace std;

int main()
{
    int myNumber = 5;
    cout << &myNumber << endl;

    return 0;
}
```



Addresses

```
0x7fff3c5b2eac

Process returned 0 (0x0)   execution tim
Press ENTER to continue.
```

Where else have we seen &?

When we pass something by reference!

Memory Addresses

Passing a parameter by-reference with the **reference operator**

```
void SetNumber( int& number )
{
    number = 5;
}
```

Outputting the address of a variable with the **address-of operator**

```
int myNumber;
cout << "Variable address: " << &myNumber << endl;
```

Take note of the different ways we can use the &.

When passing by reference, the & goes between the **data type** and the **parameter name.**

When getting the address of, we use it before the **variable name.**

Memory Addresses

Every variable has its own memory address.

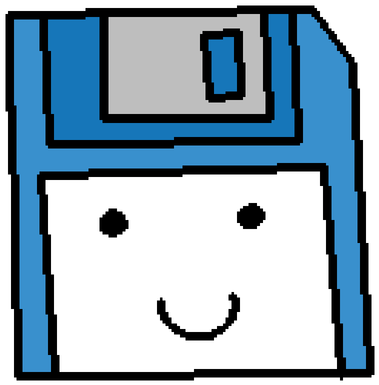
```
cout << "Variable value 1:   " << myNumber1 << endl;
cout << "Variable address 1: " << &myNumber1 << endl << endl;

cout << "Variable value 2:   " << myNumber2 << endl;
cout << "Variable address 2: " << &myNumber2 << endl << endl;

cout << "Variable value 3:   " << myNumber3 << endl;
cout << "Variable address 3: " << &myNumber3 << endl;
```

```
Variable value 1:   5
Variable address 1: 0x7fff9880b774

Variable value 2:   20
Variable address 2: 0x7fff9880b778

Variable value 3:   10
Variable address 3: 0x7fff9880b77c
```

Since we can obtain a variable's memory address, we can also store that memory address in a variable.

That's where **pointers** come in...

Pointers



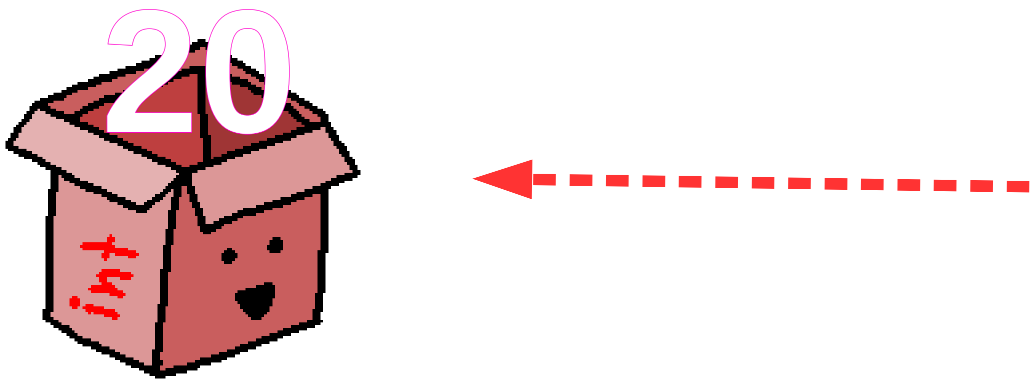
Pointers

We may want to create a variable that stores the memory address of another variable.
These variables are called **pointers**,
and are very important in C++.

A pointer variable is denoted with an asterisk, *
which does ***not*** mean multiplication here.

```
// Variables
int number = 10;
string text = "Hello!";
float money = 9.99;
```

```
// Pointer Variables
int* ptrInteger;
string* ptrString;
float* ptrFloat;
```

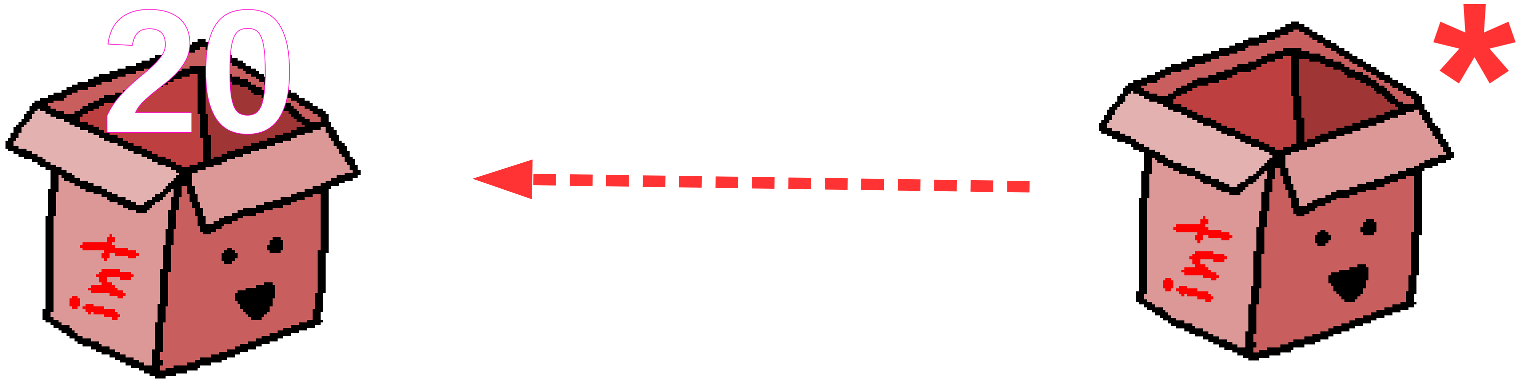


Pointers

We can assign the value of a pointer variable to the memory address of a different variable (of the same base type) by using the **address-of operator &**

Note that after we declare a pointer of some **data-type***, we don't use that asterisk when we're assigning data to the pointer.

```
// Assign memory addresses
// to pointers
ptrInteger = &number;
ptrString = &text;
ptrFloat = &money;
```



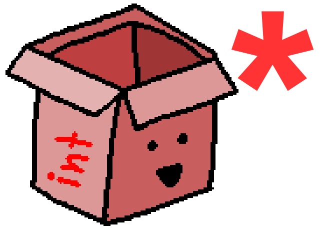
Pointers

```
// Variables
int number = 10;
string text = "Hello!";
float money = 9.99;

// Pointer Variables
int* ptrInteger;
string* ptrString;
float* ptrFloat;

// Assign memory addresses
// to pointers
ptrInteger = &number;
ptrString = &text;
ptrFloat = &money;
```

Remember:
Declare a pointer with **data-type***

Assign a pointer to an address with
&variableName

The challenging part about pointers is keeping these operators straight, so make an effort to look over it and they won't cause you too much trouble!

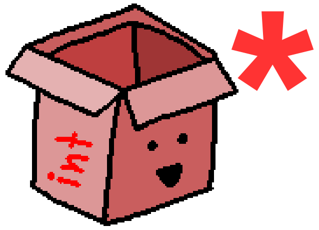


Pointers

After we've assigned our **pointers** to memory addresses,
we can view the memory address that a pointer is pointing to,
by simply accessing that pointer directly.

```
// We can also access the
// address that the pointer
// is pointing to:
cout << "ptrInteger points to: " << ptrInteger << endl;
cout << "ptrString points to:  " << ptrString << endl;
cout << "ptrFloat points to:   " << ptrFloat << endl << endl;
```

Remember:
Direct access to a pointer gives you the **memory address** of the variable it is pointing to.

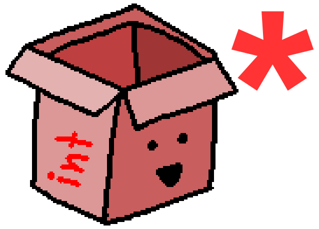


Pointers

If we want to access the **value of the pointed-to variable**,
we need to prepend * to the **pointer variable**.
In this context, it is the **de-reference operator.**

```
// Can access the value of
// those variables through
// the pointer...
cout << "Number: " << (*ptrInteger) << endl;
cout << "String: " << (*ptrString) << endl;
cout << "Float:  " << (*ptrFloat) << endl << endl;
```

Remember:
Prepending a * to a pointer gives you the **value of the pointed-to variable.**



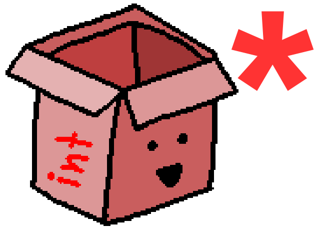
Pointers

We can use this either get the value of the pointed-to variable,
or set the value of the pointed-to variable.

```
cout << "Number: " << (*ptrInteger) << endl;
cout << "String: " << (*ptrString) << endl;
cout << "Float:  " << (*ptrFloat) << endl << endl;
```

```
*ptrInteger = 30;
*ptrString = "New Text";
*ptrFloat = 3.99;
```

Remember:
Prepending a * to a pointer gives you the **value of the pointed-to variable.**

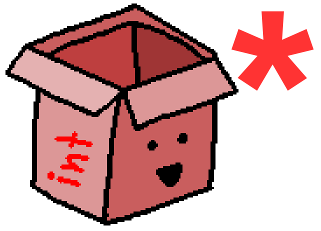


Pointers

Finally, remember that pointers are variables themselves, and also have their own memory addresses.

```
// Pointers have their own
// addresses, too, since they
// are also variables.
cout << "ptrInteger address: " << &ptrInteger << endl;
cout << "ptrString address:  " << &ptrString << endl;
cout << "ptrFloat address:   " << &ptrFloat << endl;
```

The address of the pointer is the **address of the actual pointer variable.**



Pointers

The address of the pointer is the **address of the actual pointer variable.**

```
cout << "ptrInteger address: " << &ptrInteger << endl;
```

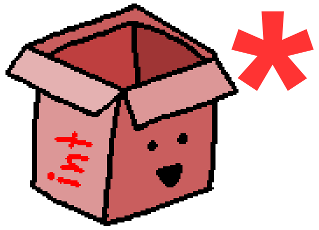
```
ptrInteger address: 0x7fff329266b8
```

The value of the pointer is the **address of the pointed-to variable.**

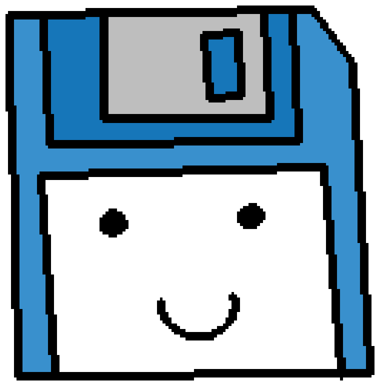
```
cout << "ptrInteger points to: " << ptrInteger << endl;
```

```
ptrInteger points to: 0x7fff329266a8
```

Keep these straight!



Time to review!



Review!

#1: Prepend **& (address-of operator)** to a variable name to get its memory address

```
int myNumber;
cout << "Variable address: " << &myNumber << endl;
```

```
0x7fff9880b774
```

#2: Pointer variables are declared by adding a ***** between the data-type and the variable name.

```
int* ptrInteger;
```

#3: We can assign a pointer variable the address of another variable (of same data-type) using the address-of operator **&**

```
ptrInteger = &number;
```

Review!

#4: Direct access to a pointer gives you the **memory address** of the variable it is pointing to.

```
cout << "ptrFloat points to:    " << ptrFloat << endl << endl;
```

```
ptrFloat points to:    0x7fff329266ac
```

#5: Prepending a *** (dereference operator)** to a pointer gives you the **value of the pointed-to variable.**

```
cout << "String: " << (*ptrString) << endl;
```

```
String: Hello!
```

Review!

#4: Direct access to a pointer gives you the **memory address** of the variable it is pointing to.

```
cout << "ptrFloat points to:    " << ptrFloat << endl << endl;
```

```
ptrFloat points to:    0x7fff329266ac
```

#5: Prepending a *** (dereference operator)** to a pointer gives you the **value of the pointed-to variable.**

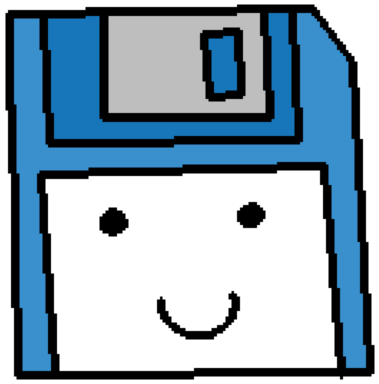
```
cout << "String: " << (*ptrString) << endl;
```

```
String: Hello!
```

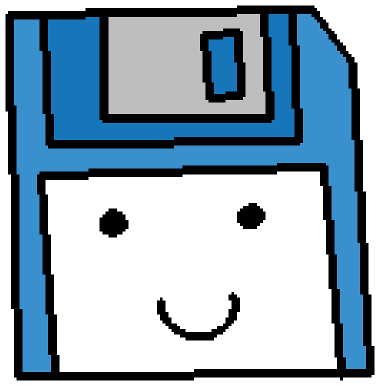
#6: Prepending a *** (dereference operator)** to a pointer
gives also will allow us to set the **value of the pointed-to variable.**

```
*ptrString = "New Text";
```

Sample Program



new and delete



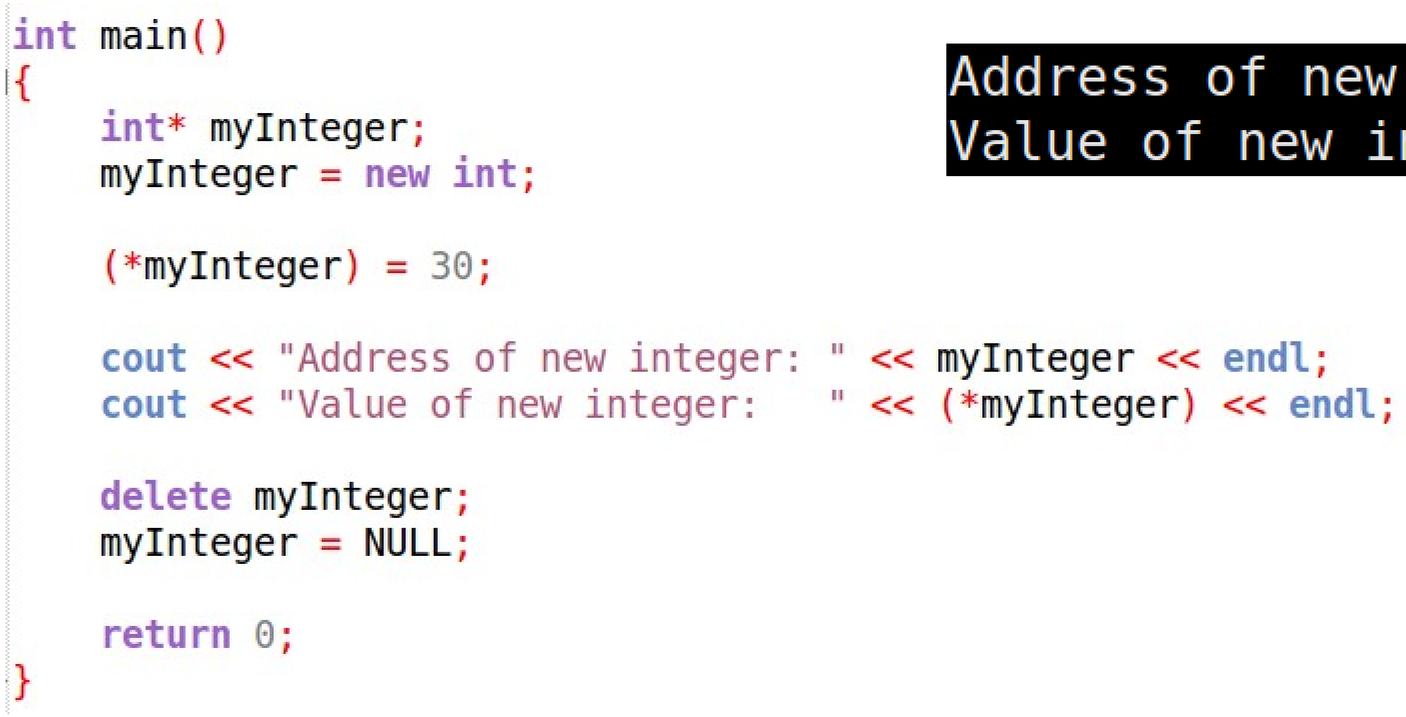
new and delete

We can use pointers to create variables as well.
This will come in handy once we're working with dynamic arrays
and building our own data structures.

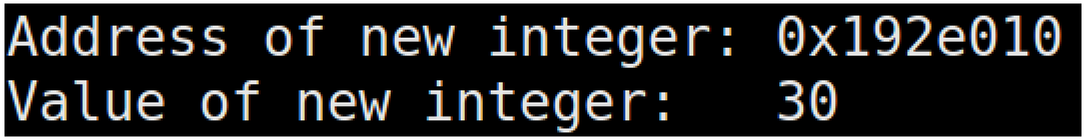
The **new** keyword allows us to **allocate memory**,
and the **delete** keyword will **free that memory.**

Therefore, any time you call <u>**new**</u>, you should also call <u>**delete**</u>.

new and delete



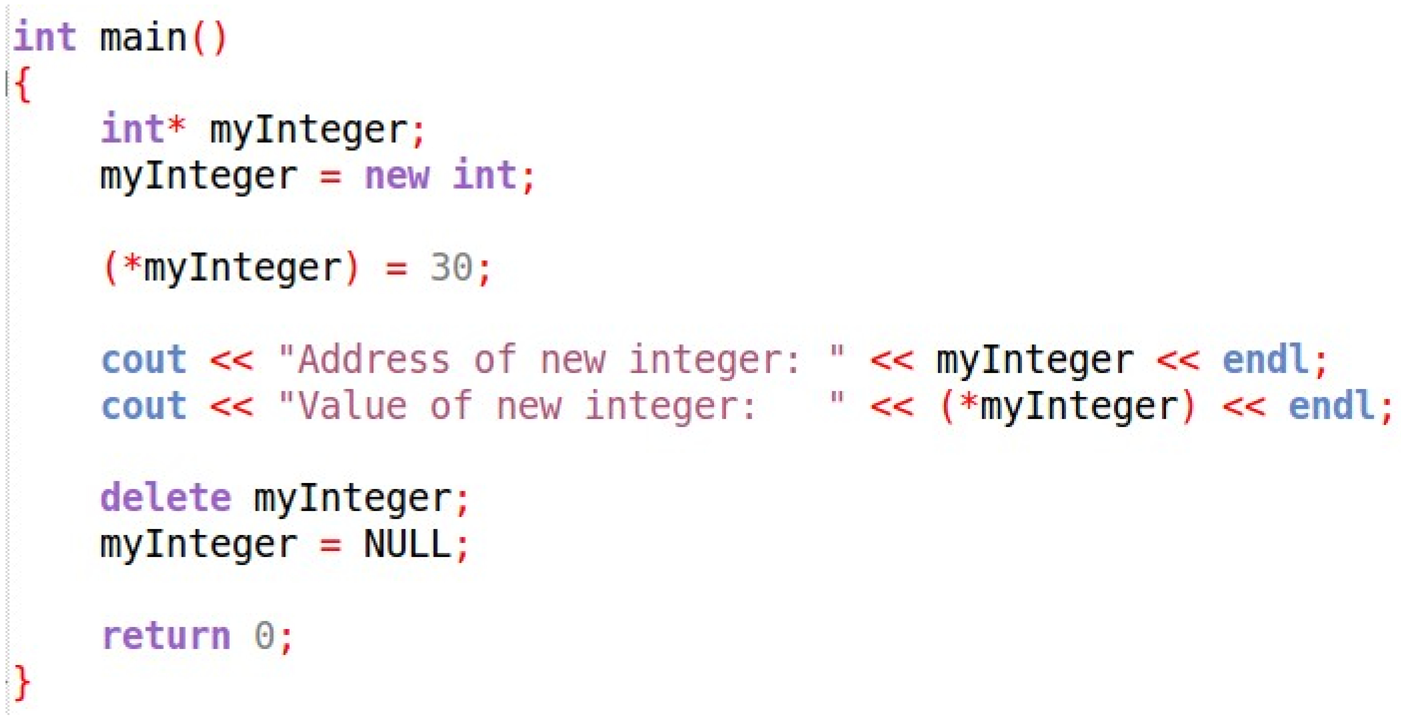
```
int main()
{
    int* myInteger;
    myInteger = new int;

    (*myInteger) = 30;

    cout << "Address of new integer: " << myInteger << endl;
    cout << "Value of new integer:   " << (*myInteger) << endl;

    delete myInteger;
    myInteger = NULL;

    return 0;
}
```



```
Address of new integer: 0x192e010
Value of new integer:   30
```

When you create a new variable in this manner, you still need to follow the rules of pointers –
you can only access the variable's **value** by using the **de-reference operator ***

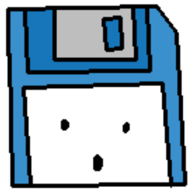
new and delete



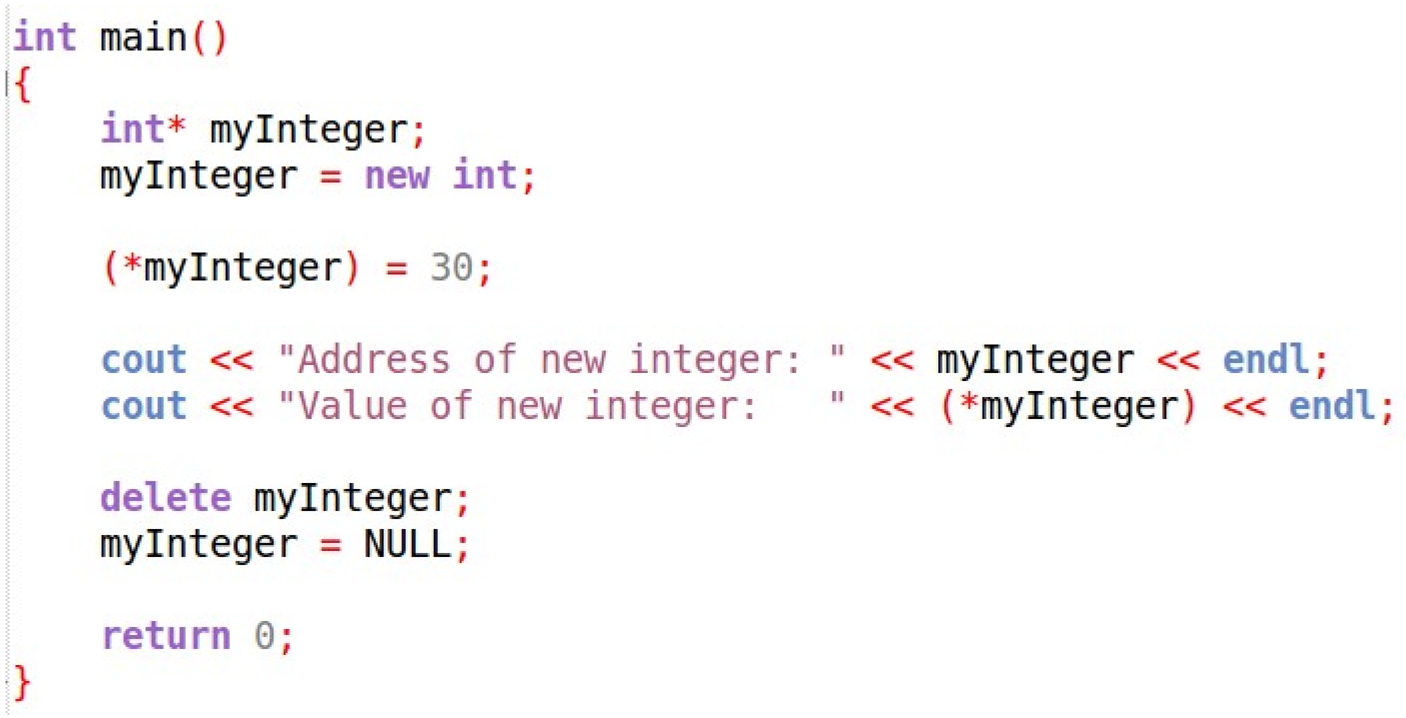
```
int main()
{
    int* myInteger;
    myInteger = new int;

    (*myInteger) = 30;

    cout << "Address of new integer: " << myInteger << endl;
    cout << "Value of new integer:   " << (*myInteger) << endl;

    delete myInteger;
    myInteger = NULL;

    return 0;
}
```

If you neglect to call **delete** to correspond to a **new**,
you will create a **memory leak.**
Each time your program is run, it will allocate memory, but never free it.

To regain the lost memory, you would
have to restart your computer.



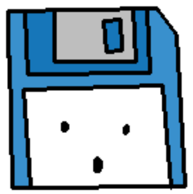
new and delete



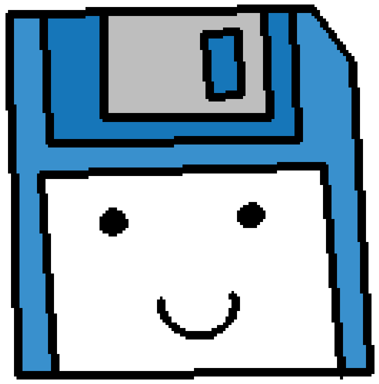
```
int main()
{
    int* myInteger;
    myInteger = new int;

    (*myInteger) = 30;

    cout << "Address of new integer: " << myInteger << endl;
    cout << "Value of new integer:   " << (*myInteger) << endl;

    delete myInteger;
    myInteger = NULL;

    return 0;
}
```

After you free the memory that a pointer is pointing to,
it is also good practice to reassign the pointer to the value of NULL.

Otherwise, it will still be pointing at a memory address,
and could cause an access error, because that memory could be reused by
another program after it has been freed by your program!!



Pointers as Parameters



Pointers as Parameters

Up until now, we've passed variables **by-reference** in order to return multiple values from a function, or to optimize our program by avoiding the copying of objects.

```
void Quadratic_Reference( float a, float b, float c, float & x1, float & x2 )
{
    x1 = ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
    x2 = ( -b - sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
}
```

```
Quadratic_Reference( a, b, c, x1, x2 );
cout << "x1 = " << x1 << ", x2 = " << x2 << endl;
```

We can do the same with pointers, by passing memory addresses:

```
void Quadratic_Pointer( float a, float b, float c, float * x1, float * x2 )
{
    *x1 = ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
    *x2 = ( -b - sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
}
```

```
Quadratic_Pointer( a, b, c, &x1, &x2 );
cout << "x1 = " << x1 << ", x2 = " << x2 << endl;
```

Pointers as Parameters

When you utilize pointers in this way, there are more steps to remember than with simply passing variables by reference.

A: Make sure your parameters are pointers.

B: Make sure to de-reference the pointer parameters when assigning values to them.

C: Make sure to pass the addresses of the variables in question using the address-of operator.

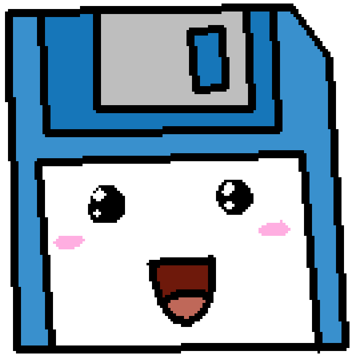
```
void Quadratic_Pointer( float a, float b, float c, float * x1, float * x2 )
{
    *x1 = ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
    *x2 = ( -b - sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
}

Quadratic_Pointer( a, b, c, &x1, &x2 );
cout << "x1 = " << x1 << ", x2 = " << x2 << endl;
```

A

B

C

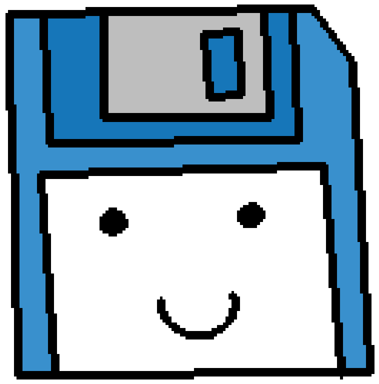
Pointers as Parameters



For the most part, it is simpler to stick to passing parameters as references.

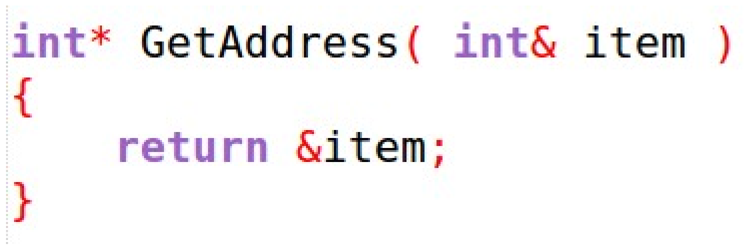
```
void Quadratic_Reference( float a, float b, float c, float & x1, float & x2 )
{
    x1 = ( -b + sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
    x2 = ( -b - sqrt( b * b - 4 * a * c ) ) / ( 2 * a );
}

Quadratic_Reference( a, b, c, x1, x2 );
cout << "x1 = " << x1 << ", x2 = " << x2 << endl;
```

Pointers as return types



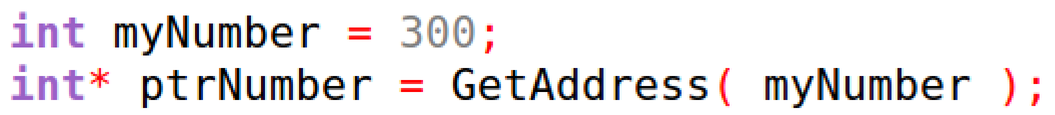
Pointers as return types

We can also return a pointer from a function,
we just need to make sure to set the return type to a pointer.



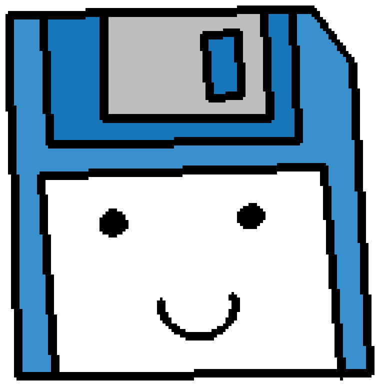
```
int* GetAddress( int& item )
{
    return &item;
}
```

We store the result in a pointer variable as well.



```
int myNumber = 300;
int* ptrNumber = GetAddress( myNumber );
```

Pointers of Objects



Pointers of Objects

We can also use pointers to point to variables that are types **objects – classes and structs.**

Our objects may have member functions and/or variables,
so if we need to use * to dereference a normal variable to get its value,
how do we dereference an object to get to its members?

Pointers of Objects

We can use the dereference operator in order to access members of an object, however, note that you will need to surround the dereference operator and the pointer name with parenthesis:

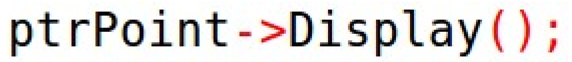
```
Point* ptrPoint = &firstPoint;
(*ptrPoint).Set( 2, 5 );
```

The following will not work, due to C++'s order of operations:

```
*ptrPoint.Display();
```

Pointers of Objects

You can also use the **member access operator ->**
as a shortcut, instead of wrapping the dereference and pointer name in parenthesis:



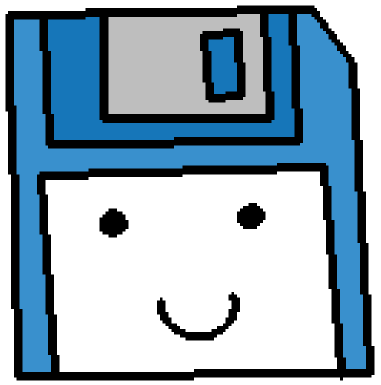
```
ptrPoint->Display();
```

You can use this with both member functions and member variables.
It is more standard to see -> used.

Pointers of Objects

```
Point* ptrPoint = &firstPoint;

cout << "\n Pointer value:" << endl;
// Option 1:    Access with the de-reference operator;
//              must be surrounded with parenthesis.
(*ptrPoint).Display();

// This won't work due to order of operations:
// *ptrPoint.Display();

cout << "\n Pointer value:" << endl;
// Option 2:    Access with the member access operator:
ptrPoint->Display();
```

Finally...



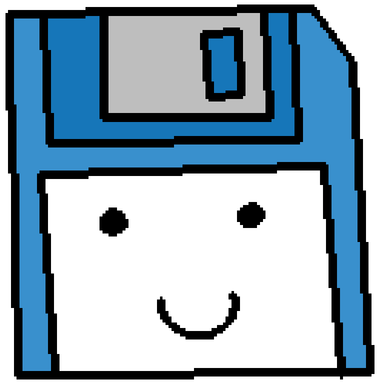
We have a lot more to cover with pointers and this is just the beginning.

Next up is memory management, and creating dynamically sized arrays.

And later on, we will be writing data structures, that contain pointers and manipulate memory.

As always, make sure to practice!

Program with Pointers



Sample Programs...

A. Students

B. Game rooms

Sample Programs...

A. Students

B. Game rooms

Sample Programs...

A. Students

B. Game rooms