C++ Programming

Pointers and Memory

one problem for heavy pointer is that
it decrease the readability of code.
we use class, vectors to take place
of pointers.

Pointers and Memory

- Pointers
- Void Pointers
- Pointer Casting
- Smart Pointers

Pointers

- Pointers in C++ function very similarly to how they work in C
- In C++, a pointer variable can point to any sort of memory
  - Pointer to basic types (like an integer)
  - Pointer to a structure
  - Pointer to an array (more on this soon)
  - Pointer to a class (more on this soon as well)
  - etc.
- This is only natural as a pointer holds the address of something in memory and everything in memory has to have an address …

Pointers

compare with malloc,
new does not require for
the size to allocate.

- There are two main uses for pointers in C++:
  - As a way of referring to memory allocated dynamically off the heap (using the `malloc()` function or the operator `new`) local param -> stack frame.
    - This allows for data that can be created on the fly or dynamically sized (such as linked lists, trees, etc.)
  - When passing large chunks of data to a function or method, passing a pointer can be more efficient, as it reduces copying and speeds up processing
    - For example, when passing a large array (and, again, we'll see more on arrays soon)

by default, c++ pass by value,
so if we're passing a huge file,
it takes a lot of memory and
time.

Pointers

But Java does ([illegible]!)

- Recall that as C++ has no built-in garbage collection, the programmer is responsible for freeing up dynamically allocated storage when they are done with it (using `free()` or `delete`, depending on how it was allocated in the first place)
- This can have performance advantages, but is cumbersome and can be painful / error-prone so it has to be done carefully
- If a programmer fails to do this properly, their program will leak memory leading to performance and stability problems (and crashes!)

Declaring Pointers

- Pointers are declared using the “*” notation, and are designated to point to certain types at the time, as in the examples below

```
int *pi;     // pi is a pointer to an integer
double *pd;  // pd is a pointer to a double
char *s;     // s is a pointer to a char
```

c++ has built-in String type, but u still could use *s if u like to do so.

Declaring Pointers

- This said, C++ also has a notion of a pointer without a specific type attached to it, which can be handy at times
- These are declared as:

```
void *vp;
```

- We will come back to these in a bit to show how they can be used

Reference (Address-of) Operator (&)

- Reference Operator &: When applied to a variable, & generates a pointer-to (or address-of) the variable
- Example:

p is a pointer to sth

```
int *p;   // p is a pointer to int (a declaration)
int c;
p = &c;   // causes p to point to c
```

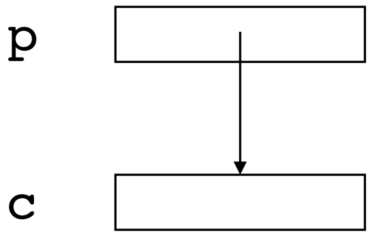
=> p is assigned to the address of c.

- Because p now has the address of and points to c, it can manipulate c indirectly!   (This can be useful, but one must exercise caution!)

Reference (Address-of) Operator (&)

```
int *p;   // p is a pointer to int (a declaration)
int c;
p = &c;   // causes p to point to c
```

at this point, p does not point to anything
we just get a space allocate for the type, but the value inside is unknown
So, it is better to declare as:
int *p = null;



Reference (Address-of) Operator (&)

```
int *p;  // suppose p is at address 1000 in memory
int c;   // and c is at address 1004 in memory
p = &c;  // stores the address of c in p as a pointer
```

1000	1004
1004	

Pointers and Assignment

- You can make a pointer point at whatever another pointer is pointing at using the standard assignment operator (=)
- Example:

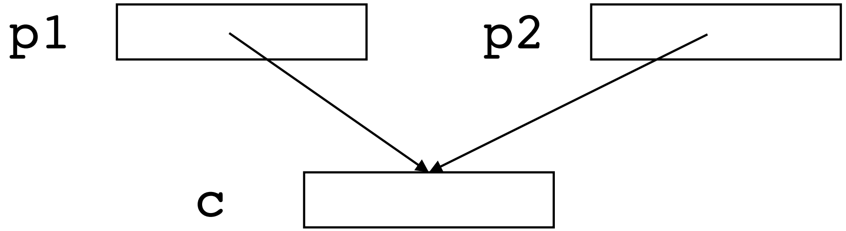
```
int *p1; // p1 is a pointer to int (a declaration)
int *p2; // p2 is a pointer to int (a declaration)
int c;
p1 = &c; // causes p1 to point to c
p2 = p1; // causes p2 to point to c as well
```

p2 is assigned to the address of c.

Pointers and Assignment

```
int *p1; // p1 is a pointer to int (a declaration)
int *p2; // p2 is a pointer to int (a declaration)
int c;
p1 = &c; // causes p1 to point to c
p2 = p1; // causes p2 to point to c
```

make a copy of p1.



Pointers and Assignment

```
int *p1;  // suppose p1 is at address 1000 in memory
int *p2;  // and p2 is at address 1004 in memory
int c;    // and c is at address 1008 in memory
p1 = &c;  // stores the address of c in p1 as a pointer
p2 = p1;  // stores the address of c in p2 as a pointer
```

1000	1008
1004	1008
1008	

Pointers and Assignment

- If you want a pointer to point at nothing, you should make it a null pointer
- This can be done by either:
  - Assigning it a value of NULL (the traditional, C style way)
  - Assigning it a value of nullptr (a more properly typed C++ way)

- Generally, if you create a pointer but won't be using it until later, it is safer to null it out on creation to avoid accidentally using an uninitialized pointer

Dereference Operator (*)

- When we want to access or manipulate what a pointer is pointing at, we dereference the pointer first. Example:

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
*p = 10; // assigns the value of 10 to variable c through p
```

- Note that if we just said:

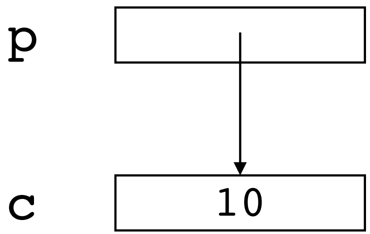
```
p = 10;
```

we would be giving p the address 10, and having it effectively point to whatever is sitting in memory at that address!

it is dangerous!

Dereference Operator (*)

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
*p = 10; // assigns the value of 10 to variable c via p
```



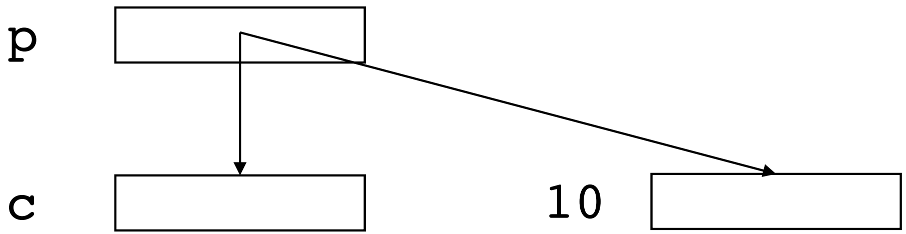
Dereference Operator (*)

```
int *p;   // suppose p is at address 1000 in memory
int c;    // and c is at address 1004 in memory
p = &c;   // stores the address of c in p as a pointer
*p = 10; // assigns the value of 10 to address 1004
```

1000	1004
1004	10

Dereference Operator (*)

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
p = 10;  // causes p to point to address 10
```



Dereference Operator (*)

```
int *p;   // suppose p is at address 1000 in memory
int c;    // and c is at address 1004 in memory
p = &c;   // stores the address of c in p as a pointer
p = 10;   // stores the address 10 in p as a pointer
```

1000	1004
1004	

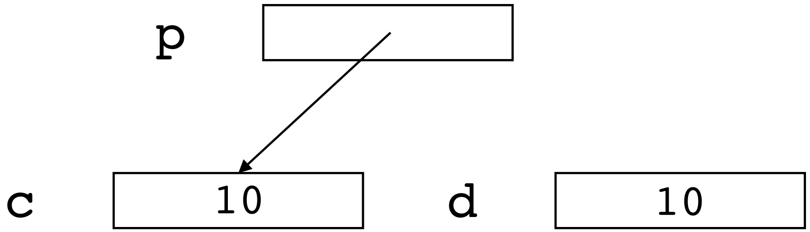
Dereference Operator (*)

- On the left hand side of an assignment, dereferencing a pointer lets you assign into the location pointed to by the pointer
- On the right hand side of an assignment, dereferencing a pointer lets you access the current value in the location it points at

```
int *p;  // p is a pointer to int (a declaration)
int c;
int d;
p = &c;  // causes p to point to c
*p = 10; // assigns the value of 10 to variable c through p
d = *p;  // assigns the value from variable c (10) to variable d
```

Dereference Operator (*)

```
int *p;  // p is a pointer to int (a declaration)
int c;
int d;
p = &c;  // causes p to point to c
*p = 10; // assigns the value of 10 to variable c via p
d = *p;  // assigns the value from c (10) to d
```



Dereference Operator (*)

```
int *p;   // suppose p is at address 1000 in memory
int c;    // and c is at address 1004 in memory
int d;    // and d is at address 1008 in memory
p = &c;   // stores the address of c in p as a pointer
*p = 10;  // assigns the value of 10 to address 1004
d = *p;   // assigns the value from address 1004 to 1008
```

1000	1004
1004	10
1008	10

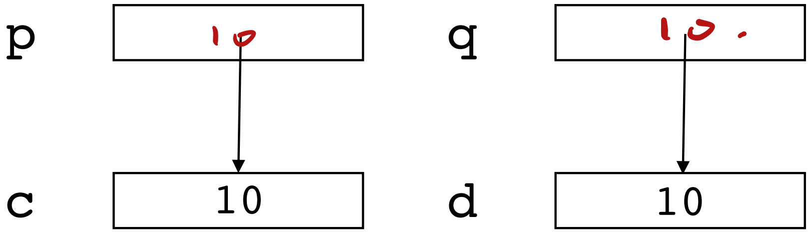
Dereference Operator (*)

- We can also dereference pointers on the left hand side and the right hand side of a single assignment statement

```
int *p, *q;  // p and q are pointers to ints
int c, d;
p = &c;      // causes p to point to c
q = &d;      // causes q to point to d.
c = 10;      // assigns the value of 10 to variable c
*q = *p;     // assigns the value from variable c (10) to variable d
             // through pointers p and q
```

Dereference Operator (*)

```
int *p, *q;   // p and q are pointers to ints
int c, d;
p = &c;       // causes p to point to c
q = &d;       // causes q to point to d
c = 10;       // assigns the value of 10 to variable c
*q = *p;      // assigns the value from c (10) to d
```



Dereference Operator (*)

```
int *p, *q;  // suppose p, q are at 1000 and 1004
int c, d;    // and c, d are at 1008 and 1012
p = &c;      // stores the address of c in p
q = &d;      // stores the address of d in q
c = 10;      // assigns the value of 10 to address 1008
*q = *p;     // assigns the value from 1008 to 1012
```

1000	1008
1004	1012
1008	10
1012	10

Dereference Operator (*)

- One must exercise care when dereferencing a pointer or else various forms of badness can happen. This includes:
- Dereferencing a pointer that hasn't been told to point at something yet:

```
int *p;   // p is a pointer to int (a declaration)
*p = 10; // assigns the value of 10 to what? *Boom!*
```

- Dereferencing a NULL pointer:

```
int *p = NULL;   // p is an integer pointer with a value of NULL
*p = 10;         // assigns the value of 10 to what? *Boom!*
```

- Dereferencing a pointer that points at inaccessible memory (e.g. by doing the bad assignment `p = 10;`)

A Quick Note on Our Pointer Examples

- This is one of those times when we’re showing something as an example that you typically wouldn’t do in practice
- The code we’ve been using for the last few slides:

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c.
```

is valid code, but pointers should be mainly used for dynamic storage (from the heap) and not generally used to point to local data (from the stack) within a function (like `c` in the above code)

A Quick Note on Our Pointer Examples

- Why?
  - We now have two ways of referring to the same location or variable in memory, and this can lead to confusion and difficulties in understanding and maintaining the code
  - If you were to return the pointer outside of the function or method containing this code and tried to use it, all kinds of bad things could happen as the stack frame is destroyed and the pointer no longer points at what you think it should

New and Delete

- As noted earlier, `new` and `delete` are the improved way of heap allocating memory in C++ (compared to `malloc` and `free` from C)
- Instead of having to calculate the number of bytes to allocate as was necessary in C, you can now simply indicate the type of thing you want and C++ will allocate one for you off of the heap, setting aside the right amount of memory in the process
- C++ will also help in initializing the created variables, setting initial values or calling constructors as necessary (in the case of objects – more on this later)

New and Delete – An Example

everything comes from heap.

```
#include<iostream>

void my_func() {
    int *p = nullptr;                       // Create a pointer and make it null
    p = new int;                            // Allocate a new integer and have pointer point at it
    *p = 1;                                 // Set our new integer to have a value of 1
    std::cout << *p << std::endl;           // Print out the value of our integer
    delete p;                               // Deallocate our integer variable and return
}
int main() {
    my_func();                              // Call our new function
}
```

memory is allocated automatically, no need of malloc operation.

New and Delete – An Example

type.

```
int *p = nullptr;
p = new int;
*p = 1;
std::cout << *p << std::endl;
delete p;
```

it is not *p = Null.
it is a type; a pointer pointing at null.
*p = Null; type = Null;
nullptr != NULL

, 1 is also remove from heap;
technically 1 is still there, but it is available.

p [null] → [1]

Stack   Heap

New and Delete – An Example

- A few notes on this example:
  - Our pointer, when declared, contains an undefined value and we should null it out ourselves manually for safety first
  - The variable allocated by `new` can be initialized by passing a parameter into it. In other words,

```
p = new int;
*p = 1;
```

is equivalent to:

```
p = new int(1);
```

New and Delete – An Example

- A few notes on this example:
  - Deleting our allocated variable does not change the value of its pointer. The pointer in fact still points to the same location on the heap, and can still be used! This is dangerous though, as using this stale pointer can lead to unexpected (bad) results. As a result, it is safest to null a pointer out after using `delete` on it, if it is going to persist longer.

    p= nullptr
  - If you were to exit the function without calling `delete`, the variable allocated on the heap would persist, resulting in a memory leak. Too many leaks over time will exhaust memory and cause your program to crash.

New and Delete – An Example

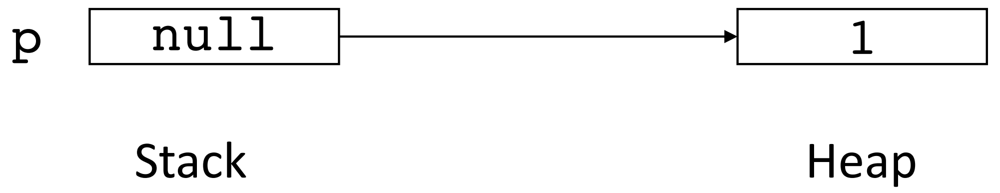
```
#include<iostream>

void my_func() {
    int *p = nullptr;                  // Create a pointer and make it null
    p = new int;                       // Allocate a new integer and have pointer point at it
    *p = 1;                            // Set our new integer to have a value of 1
    std::cout << *p << std::endl;      // Print out the value of our integer
}
int main() {
    my_func();                         // Call our new function.  Note that it returns
}                                      // without deleting its allocated memory, causing a
                                       // memory leak.  It's not terrible in this case as the
                                       // program is about to end, but is still a bad thing.
```

↑ when your program ends, everything on heap gets destroyed.

New and Delete – An Example

```
int *p = nullptr;
p = new int;
*p = 1;
std::cout << *p << std::endl;
```



New and Arrays

- A bit of care needs to be exercised using new with arrays, or more accurately, when deleting them when we are done with them
- Suppose you allocate an array of five integers like this:

```
int *array = new int[5];
```

<- try not to use array unless necessary, try vector instead.

- What were to happen if you deleted this array using:

```
delete array;   // Is this okay?
```

recursively delete.

only the head of the array.

it takes array as an integer.

New and Arrays

- This gets a bit messy as the compiler, given just an `int *` pointer, does not know if this is a pointer to a single integer or an array of them
- As a result, `delete` might not clean things up completely and memory could leak as a result
- For this reason, most compilers will issue a warning if it catches you doing this sort of thing
- Unfortunately, as a pointer is used or passed around from function to function, the compiler quickly loses the ability to warn you about such things and so you cannot rely on these warnings alone!

New and Arrays

- The bottom line is that your use of [] should match up when using new and delete for allocation
  - Anything created with `new` should be deleted with `delete`
  - Anything created with `new[]` should deleted with `delete[]`
- So, correcting our code from a couple slides back, the proper way to clean up our allocated array is using:

```
delete[] array;  // The proper way to do things!
```

Void Pointers

- As noted earlier, C++ allows pointers without a specific data type associated with them
- These pointers are referred to as void pointers
- Void pointers are frequently used in C to create generic methods as they are not tied to particular data types
- When C++ code interacts with C code however (through the use of libraries, for example), it becomes important to know a few things about them

Void Pointers

- Some key things with void pointers:
  - Void pointers cannot be dereferenced (which makes sense since they have no type and so what would the dereferenced thing be?)
  - Pointer arithmetic with void pointers is generally not permitted in the standards (which again makes sense because without a type, how do you know how big the thing is that is being pointed at?)
  - Some compilers permit pointer arithmetic with void pointers, however, assuming the size of void is 1 and so you are essentially working with a block of byte data
  - In C++, doing much work with void pointers inherently is going to require casting to and from other pointers that have known data types associated with them

Pointer Casting

- Casting is a conversion process explicitly indicated by the programmer to change data from one type to another
- This includes pointers, allowing us to convert to and from void pointers as in:

but we have to clearly know what we doing!

```
int i = 0;                  // i is an integer
void *p = (void *) &i;      // &i is an int * and so we must cast it
                            // to get a void *
int *j = (int *) p;         // p is a void * and so we must cast it
                            // again to get back a usable int *
```

Pointer Casting

- You must be careful in casting, because the compiler will trust you and assume you know what you're doing
- For example, the compiler will find the following code to be perfectly acceptable as well:

```
int i = 0;                // i is an integer
void *p = (void *) &i;    // &i is an int * and so we must cast it
                          // to get a void *
char *s = (char *) p;     // hilarity ensues
```

Pointer Casting

- Over the years, the C++ standard has evolved to include many different types of casting
- This includes static_cast, const_cast, reinterpret_cast, dynamic_cast and more, including a few variants of these
- We won’t worry about these here, but you are welcome to consult your favourite C++ reference for more!

Smart Pointers

- There have been several propositions aimed at including smart pointers in C++ over the years
- The idea is to have pointers that automatically clean up after themselves when the data being pointed at is no longer needed or cannot be used any more (as there are no longer any valid pointers pointing at the data for instance)
- The goal is not to introduce a garbage collection mechanism ala Java, but rather to provide a lightweight mechanism around some of the tediousness of pointer management with new and delete

Smart Pointers

- The mechanism we are exploring was introduced in C++14, though parts were available in earlier standards
- In this case, there are two types of smart pointers:
  - Shared pointers (shared_ptr)
  - Unique pointers (unique_ptr)

- Let's take a quick look at both types …

Shared Pointers

- Shared pointers allow multiple smart pointers to all point at the same variable, and so they “share” ownership of that variable
- They work by maintaining a reference count tracking the number of references to that variable across the shared pointers pointing to it
- If the number of references is reduced to zero, by destroying the pointers or assigning them to point at something else, the variable is automatically deleted

Shared Pointers – An Example

```
#include<iostream>

void my_func() {
    std::shared_ptr<int> sp = nullptr;   // Create a shared pointer and make it null
    sp = std::make_shared<int>();        // Allocate a new integer and have pointer point at it
    *sp = 1;                             // Set our new integer to have a value of 1
    std::cout << *sp << std::endl;       // Print out the value of our integer
}

int main() {
    my_func();                           // Call our new function.  Note that when my_func()
}                                        // returns, sp will be destroyed.  As the only
                                         // reference to our integer is gone, it will be deleted.
```

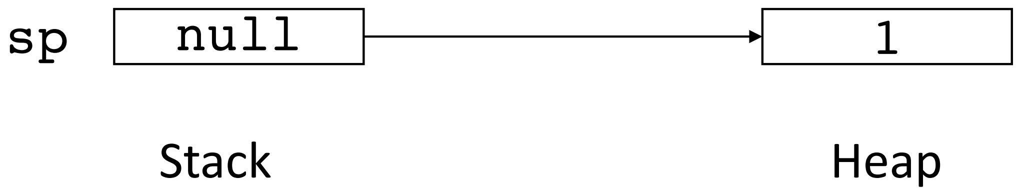
it lives on stack,

when the function ends, things on stack is automatically destroied.

Shared Pointers – An Example

after the function end, both stack and heap get destroyed.

```
std::shared_ptr<int> sp = nullptr;
sp = std::make_shared<int>();
*sp = 1;
std::cout << *sp << std::endl;
```



Shared Pointers – An Example

- A couple of notes on this example:
  - Shared pointers are traditionally initialized to be null by the compiler when declared if they are not assigned something on creation
  - The variable allocated by `make_shared` can be initialized by passing appropriate parameters into its function call. In other words,

```
sp = std::make_shared<int>();
*sp = 1;
```

null initialization,

is equivalent to:

```
sp = std::make_shared<int>(1);
```

regular point never be initialized to null but share pointer is by default.

Shared Pointers – Another Example

```
#include<iostream>

void my_func() {
   std::shared_ptr<int> sp;              // Create a shared pointer (it will default to null)
   sp = std::make_shared<int>(1);        // Allocate a new integer and have pointer point at it
   sp = std::make_shared<int>(2);        // Allocate and point at a second integer
   std::cout << *sp << std::endl;        // Print out the value of our latest integer
}
```

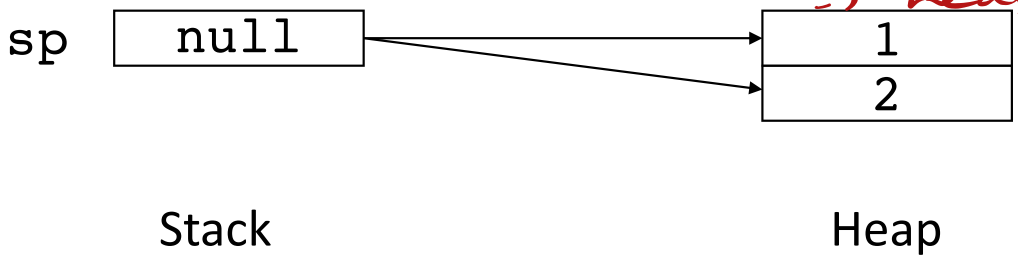
pointing at something else,
the first int is destroyed as ref count is 0

```
int main() {
   my_func();                            // Call our new function.  Note that when my_func()
}                                        // returns, sp will be destroyed.  As the only
                                         // reference to our integer is gone, it will be deleted.
```

Shared Pointers – Another Example

```
std::shared_ptr<int> sp;
sp = std::make_shared<int>(1);
sp = std::make_shared<int>(2);
std::cout << *sp << std::endl;
```

Function end
=> cleaning stack
=> sp destoried
=> reference count to
heap falls to 0
=> heap destoried.



No manually
delet needed.

Shared Pointers – One More Example

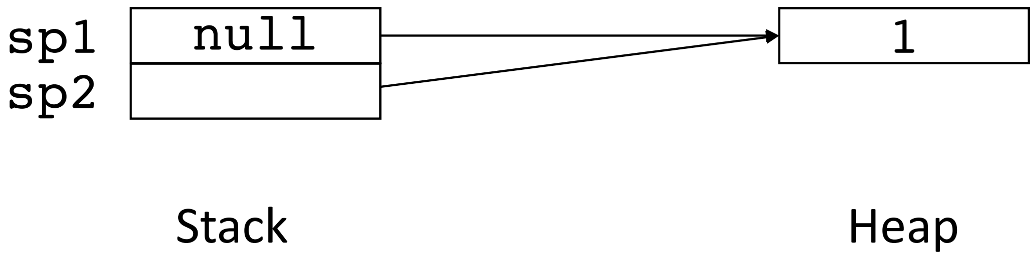
```
#include<iostream>

void my_func() {
   std::shared_ptr<int> sp1;              // Create a shared pointer (it will default to null)
   sp1 = std::make_shared<int>(1);        // Allocate a new integer and have pointer point at it
   std::shared_ptr<int> sp2 = sp1;        // Create a second shared pointer to point at this
   std::cout << *sp2 << std::endl;        // Print things out using our latest pointer
}

int main() {
   my_func();                             // Call our new function.  Note that when my_func()
}                                         // returns, sp1 and sp2 will be destroyed.  As the
                                          // references to our integer is gone, it will be deleted.
```

Shared Pointers – One More Example

```
std::shared_ptr<int> sp1;
sp1 = std::make_shared<int>(1);
std::shared_ptr<int> sp2 = sp1;
std::cout << *sp2 << std::endl;
```



Unique Pointers

- Unlike shared pointers, unique pointers allow only a single smart pointer to point at the same variable, and so ownership of that variable can be considered to be “unique”
- As only a single unique pointer can point at its variable, reference counting is not necessary; instead, as soon as a unique pointer goes out of scope, is destroyed, or has something else assigned to it, it can delete its owned variable
- In many ways, they are similar in use to shared pointers, but have additional restrictions as we will soon see

Unique Pointers – An Example

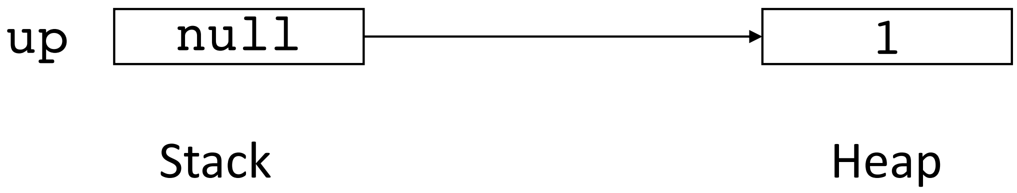
```
#include<iostream>

void my_func() {
   std::unique_ptr<int> up = nullptr;  // Create a unique pointer and make it null
   up = std::make_unique<int>();       // Allocate a new integer and have pointer point at it
   *up = 1;                            // Set our new integer to have a value of 1
   std::cout << *up << std::endl;      // Print out the value of our integer
}

int main() {
   my_func();                          // Call our new function.  Note that when my_func()
}                                      // returns, up will be destroyed.  As the only
                                       // reference to our integer is gone, it will be deleted.
```

Unique Pointers – An Example

```
std::unique_ptr<int> up = nullptr;
up = std::make_unique<int>();
*up = 1;
std::cout << *up << std::endl;
```



Unique Pointers: Similarities and Differences

- Some similarities with shared pointers:
  - Like shared pointers, unique pointers traditionally default to a value of null
  - Variables can also be initialized as they were with shared pointers
- Some differences between shared and unique pointers:
  - The compiler will make sure that there can never be more than one unique pointer pointing at a variable (otherwise it won't be unique)
  - Things that would create multiple copies of a pointer are restricted, including pointer assignment and using unique pointers as function parameters
  - A `move` function, however, allows ownership of a variable to be moved from one unique pointer to another

Unique Pointers: Examples of Restrictions

- For example, this is not allowed with unique pointers:

```
std::unique_ptr<int> up1 = std::make_unique<int>();
std::unique_ptr<int> up2 = up1;
```

X

as it would violate the uniqueness criteria by duplicating things.

- Instead, you would have to move the pointer:

```
std::unique_ptr<int> up1 = std::make_unique<int>();
std::unique_ptr<int> up2 = std::move(up1);
```

This would also set up1 to null as part of the move operation.

replace -

Unique Pointers: Examples of Restrictions

- As another example, this is also not allowed with unique pointers:

```
std::unique_ptr<int> up = std::make_unique<int>();
my_func(up);
```

  as it would violate the uniqueness criteria by copying our `up` pointer into the function on the function call.

- This, however, is fair as it does not leave a copy behind in the function that called `my_func` in the first place:

```
my_func(std::make_unique<int>());
```

Smart Pointers: A Summary

- Like most things added to C++ along the way, smart pointers are not without their controversary
- One side believes smart pointers introduce more complex syntactic baggage and that programmers should make due with `new` and `delete`, learning to manage memory properly on their own
- The other side feels that the reduction in memory mismanagement and memory leaks from the use of smart pointers is well worth it, and given the persistent memory issues arising from C and C++ code in the wild, it is difficult to argue against this