Introduction to Dynamic Memory

Preface

Because of the long history of C++, including its roots beginning with the C language, as with other aspects of the language, dynamic memory is multi-faceted. While this series of notes provides a reasonably detailed look into dynamic memory, there are details not captured. The purpose of these notes is to give a detailed enough understanding of dynamic memory in C++ that you are well prepared to learn and understand additional details on your own.

Review – Stack & Heap

It is worth reviewing the concepts of the stack and heap, because they are essential to understanding how dynamic memory works in C++.

- **Stack** This is a concept provided by the C++ language, it is created by the compiler/linker as it generates the instructions for the executable code. A stack is a growable/shinkable data structure used for temporary storage. Primarily it is used for variables and function/method parameters. Variables and parameters are valid within a certain scope. When a scope opens, memory on the stack is reserved for the variables and parameters (if any). When the scope ends, the stack shrinks and removes the memory reservation for the variables and parameters.
- **Heap** The operating system manages all memory on the computer. Typically a large section of memory is reserved for what is called the heap. The heap is used to provide memory to process that make requests for dynamic memory. Any data storage that needs to exist beyond the lifetime of any particular language scope is allocated from the heap. When requested, the OS finds an available location in the heap, marks it as used, and returns the location of that memory to the requesting program (see next section for Java References). When the requesting program no longer needs the memory, it informs the OS, allowing it to mark the location as available.

* Discuss value types and the stack with C++ * In Java string is a reference to a string. In C++ `std::string` is a stack allocated value as far as use is concerned, even if internally there is a heap allocated component.

C++ Pointers and Java References

It is important to understand some key differences between C++ and Java with respect to dynamic memory. As presented earlier, C++ uses the term ***reference*** as a mechanism to create an alias for a variable. When requesting dynamic memory in Java, the `new` keyword is used, followed by the data type of storage to create. Upon successful allocation, a ***reference*** to the memory location is used. A C++ reference is not the same as a Java reference. *They are the same word, but have very different meanings.*

A Java reference should be thought of as an identifier or lookup number used by the Java Virtual Machine (JVM). When a Java program performs a dynamic memory allocation, the JVM, in turn, makes a request to the OS for the memory storage. The OS returns to the JVM the actual memory location of the storage. The JVM then adds this actual storage location to an internal lookup table and gives it a unique identifier. This unique identifier is returned to the Java program, it is what is called a reference. In other words, the identifier is used as a reference into a lookup table that holds the actual memory storage location for the data. When the JVM Garbage Collector (GC) determines a reference is no longer in use, it makes calls to the OS to return the requested memory and removes the reference from the internal JVM lookup table. In this way, Java is able to provide memory safety by verifying the reference actually exists and that any attempt to use memory it references is valid.

** Draw Diagram **

C++ uses the same `new` keyword (it is an operator in C++, this becomes important eventually) to perform a dynamic memory request. This request is translated into a direct call to the OS, which results in the actual memory location for the storage being returned to the C++ program. The C++ data type for these storage locations is called a pointer. For every data type in C++, there is a corresponding pointer data type, it is formed by adding an asterisk `*` to the data type. The following declarations illustrate:

```
int* a = new int(1);
double* b = new double(3.14159);
std::string* c = new std::string("This string is dynamically allocated");
```

Notice the addition of the `*` along with the data type, this is what defines a pointer. Any data type in C++ can be turned into a pointer type simply by adding the `*` symbol. A pointer declaration is a data type, therefore, it is possible to declare a pointer to a pointer, like the following:

```
int** a;
```

What is the purpose of doing something like this and how would the dynamic allocation be performed? Stay tuned.

I will refer to these pointer types as *raw pointers*. Right after the section on raw pointers, the topic of another type of pointers is presented, *smart pointers*. In these notes, the terms *pointer* or *pointers* are used, referring to either raw pointers or smart pointers as the context indicates.

Note: I don't have any recommendation an `std::string` is dynamically allocated as shown above, only showing it as something that can be done.

Arrays & Pointers

The name of a raw array and raw pointers have a special relationship. The name of an array is a pointer to the first array element location. If we have the following array declaration:

```
int primes[] = { 2, 3, 5, 7 };
```

The value stored in the primes variable is the actual memory storage location of the first element in the array. So what? So this...

```
int primes[] = { 2, 3, 5, 7 };
int* pointerToPrimes = primes;

std::cout << primes[0] << " : " << pointerToPrimes[0] << std::endl;
std::cout << primes[1] << " : " << pointerToPrimes[1] << std::endl;
std::cout << primes[2] << " : " << pointerToPrimes[2] << std::endl;
std::cout << primes[3] << " : " << pointerToPrimes[3] << std::endl;
```

The second line of code takes the value stored in `primes` and places it into the variable `pointerToPrimes`. Because these are compatible data types, no compiler warning or error is generated. Because of the special relationship between the names of arrays and pointer types, a pointer type can be indexed just like an array. The code sample above demonstrates that both `primes` and `pointerToPrimes` report the same values when indexed as arrays.

This relationship is only true of raw arrays, it does not apply to `std::array` or `std::vector`.

This relationship is an excellent segue into dynamic allocation of raw arrays. Consider the following code:

```
int primes[] = { 2, 3, 5, 7 };
int* dynamicPrimes = new int[4];

dynamicPrimes[0] = 2;
dynamicPrimes[1] = 3;
dynamicPrimes[2] = 5;
dynamicPrimes[3] = 7;
```

The first line of code is what we've been doing all along. The second line of code declares a pointer to an integer and allocates an array of integers. Once the array is allocated, it is then used just like any other array.

Rather than allocate and then assign, it is possible to allocate and initialize in a single statement:

```
int* dynamicPrimes2 = new int[4]{ 2, 3, 5, 7 };
```

Pointers & Reference Memory Responsibility

The JVM periodically invokes the GC, during which time it releases the memory associated with any unused references. When this happens, the GC makes a call to the OS, releasing the memory. This eliminates the need for the programmer to write code to tell the JVM directly when it is time to release the memory.

** Draw diagram **

There is no similar concept of the JVM in C++. Instead, the executable produced by the C++ compiler/linker is complete in and of itself, not requiring an additional execution context like the JVM. In a C++ program, when a pointer goes out of scope or loses track of allocated memory, that memory is still associated with the process. Once lost, there is no way for the program to rediscover that memory location. This is known as a *memory leak*. The memory remains associated with the process until it is terminated, at which time the OS can reclaim the memory.

When working with raw pointers, it is the responsibility of the programmer to write code that releases memory when it is no longer needed. Use the `delete` keyword (operator) to release memory when it is no longer needed. The following code demonstrates:

```
int* primes = new int[4];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
… do something with the prime numbers…
delete []primes;
```

Unfortunate C++ deal: Notice the use of the array brackets `[]` in the `delete []primes;` statement. When deleting dynamically allocated arrays, this is necessary to let C++ know an array is being deleted, versus a single value. Only for array deletion are these necessary, all other deletes of dynamic memory do not have the array brackets.

After the memory is freed, the value stored in the `primes` variable remains unchanged, but it is no longer valid. It is a memory location, but that memory location is no longer associated with the process. Any attempt to access the memory stored at that location might result in the process being terminated by the OS, because it is an illegal memory access.

Following the release of memory, if the pointer variable isn't immediately going out of scope, it is recommended to set it to the value `nullptr`, indicating it points to nothing.

```
primes = nullptr;
```

Arrays – Stack or Heap?

Consider the following array declaration and initialization in Java:

```
int[] primes = { 2, 3, 5, 7 };
```

All arrays in Java are heap allocated, therefore, this array is allocated from the heap. It is relevant to note the variable `prime` exists on the stack, but the array it references is allocated from the heap.

Let's go back to the following C++ code and consider it in more detail:

```
int primes[] = { 2, 3, 5, 7 };
int* dynamicPrimes = new int[4];
```

The variables `primes` and `dynamicPrimes` exist on the stack, but what about they arrays associated with them?

We know for sure the second statement is allocating the array from the heap. The variable `dynamicPrimes` is located on the stack, but the memory it points to is allocated from the heap.

We know the variable `primes` is also a pointer to the array data. Because it is a pointer, does that mean the array data it points to is heap allocated? The answer is no. A pointer can hold a memory location anywhere in memory, that can be a memory location from the stack or heap.

The first array exists completely on the stack. Both the variable `primes` and the array data itself exist on the stack. When the scope for the `primes` array ends, both the `primes` variable and the array data cease to exist.

Consider the following function and its use (valid technique):

```
int* generate10Primes()
{
    int* primes = new int[10];
    primes[0] = 2;

    for (int i = 1; i < 10; i++)
    {
        primes[i] = getNextPrime(primes[i - 1]);
    }

    return primes;
}

...

int* tenPrimes = generate10Primes();
```

** Draw diagram of the stack and heap allocation **

The variable `primes` is located on the stack. The array it points to is dynamically allocated from the heap. Because the data type of `primes` is a pointer type, it can be indexed using array brackets. The return statement returns a copy of the pointer which is stored in the `tenPrimes` variable. After the scope of the function is closed, following the return, the `primes` variable no longer exists, but the calling code has stored a copy of the memory location in `tenPrimes`, keeping track of the dynamically allocated data. All is well!

Let's rewrite the function as follows (invalid technique):

```
int* generate10Primes()
{
    int primes[10];
    primes[0] = 1;

    for (int i = 1; i < 10; i++)
    {
        primes[i] = getNextPrime(primes[i - 1]);
    }

    return primes;
}

...

int* tenPrimes = generate10Primes();
```

** Draw diagram of the stack **

In this case, the variable `primes` ***and*** the array elements are located on the stack; `primes` is still a pointer to the first element in the array. Other than that, the rest of the code is the same as the previous implementation. While this code compiles (may or may not generate a warning depending on the compiler and warnings settings), there is a big problem. The return statement returns the value stored in `primes`, which is a pointer to the location of the array data, just as in the first version of the function, along with the memory location being copied into `tenPrimes`; nothing technically incorrect there. The problem is when the function scope ends, following the return, the data on the array is no longer accessible/valid, because the stack shrinks and removes the space used for the array elements. The `tenPrimes` variable has a pointer to the location on the stack, but it is no longer a valid location for the array data because it has been reclaimed by the stack. W*hile C++ allows this code, because of the relationship between array names and pointers, it is a mistake.*

Using Pointer Variables

We've seen how to declare a pointer type, along with dynamically allocating simple types and arrays. For arrays, the array index operator can be used to access the elements. What about a pointer to a simple type, or primitive? Or how about obtaining the memory address of a variable. Let's take a look.

```
int a = 10;
int* ptrA = &a;

std::cout << ptrA << " : " << *ptrA << std::endl;
*ptrA = 20;
std::cout << a << " : " << *ptrA << std::endl;
```

The second line of code show the use of the *address-of* `&` operator. The address-of operator returns the location in memory of the variable immediately following it. In this case, `&a` returns the location in memory for the `a` variable, which is then stored in the `ptrA` variable. At this point we say that "ptrA points to a".

** Draw the diagram for these two variables **

The address-of operator is easily confused with a reference type. They are not the same, even though the symbol is the same, they are very different things; context matters.

The third line of code shows two uses of the `ptrA` variable. The first use prints out the value stored in `ptrA`, a memory location. The second use has the an asterisk `*` in front of it, in this context it is known as the *dereference* operator. This operator returns the value stored at the memory location of the pointer type it is placed in front of. We say that it "dereferences the pointer".

The fourth line of code shows how to change the value stored at the memory location stored in `ptrA`. The dereference operator is necessary to do this. The value stored in `ptrA` stays the same, a memory location, but the value at that memory location is changed to 20. When the value is changed through the use of the pointer, it is changing the value stored in the `a` variable.

Finally, the last line of code demonstrates that the value stored in variable `a` has been changed through the use of the pointer variable `ptrA`.

When run on my computer, this code outputs the following:

```
0x7ffdc2ab47b4 : 10
20 : 20
```

The first value is a memory location and will probably be different each time the program is run.

Raw Pointer Arithmetic

Let’s go back to the array of prime integers we declared earlier and take another look at pointers and what I like to call “new math”.

```
int primes[] = { 2, 3, 5, 7 };

std::cout << *(primes + 0) << std::endl;
std::cout << *(primes + 1) << std::endl;
std::cout << *(primes + 2) << std::endl;
std::cout << *(primes + 3) << std::endl;
```

This code uses pointer arithmetic and pointer dereferencing to print out the values in the primes array, rather than using the array bracket operator to access the elements. Any pointer can be used in this way, this code happens to show it through the use of an array name.

Let’s add some more code to this example:

```
std::cout << (primes + 0) << std::endl;
std::cout << (primes + 1) << std::endl;
std::cout << (primes + 2) << std::endl;
std::cout << (primes + 3) << std::endl;
```

The output from this additional code is (on my computer):

```
0x7ffcfbbc5f60
0x7ffcfbbc5f64
0x7ffcfbbc5f68
0x7ffcfbbc5f6c
```

These values are memory locations. Note how the difference in the values is 4 between each location instead of the 1 you might expect based on looking at the code. The reason for this is that when C++ performs pointer arithmetic, the value added is the size (in bytes) of the type pointed to multiplied by the value being added. In the first statement it is 0 multiplied by 4 (the size in bytes of an int on my computer), the second is 1 multiplied by 4, the third 2 multiplied by 4, and the fourth 3 multiplied by 4.

If we change the data type to a `short int`, the output is:

```
0x7ffcfbbc5f68
0x7ffcfbbc5f6a
0x7ffcfbbc5f6c
0x7ffcfbbc5f6e
```

This time the difference in each memory location is 2 (bytes), because the size of a `short int` is 2 bytes.

Pointers As Function Parameters

As already discussed, C++ provides pass-by-value as the default means for passing function parameters. It also has a pass-by-reference capability through the use of reference variables. The type of a parameter can be a pointer, and that pointer can be passed by value and used to change the data in the calling variables, similar to a variable reference.

Remember the swap function we wrote using pass-by-reference. Let’s take a look at a similar function using pointers and pass-by-value.

```
void swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}


int a = 1;
int b = 2;

std::cout << "Before swap: a = " << a << ", b = " << b << std::endl;
swap(&a, &b);
std::cout << "After swap:  a = " << a << ", b = " << b << std::endl;
```

** Draw diagram of these variables and the calling variables **

The two parameters to this function are pointer types. In order to change and use the values pointed to by these variables, the dereference operator is used in the function. Notice also, the call to the `swap` function requires the address operator is used to obtain the memory address of the `a` and `b` variables.

It is worth mentioning, but not going to be covered here, pointers can be passed by reference, allowing the value of the pointers themselves to be changed!

(TODO) Constant Pointers

** Write this topic up at some point in the future, but not critical for now

Introduction to Smart Pointers

**** Must wait to do this section until class destructors are introduced ****

The Java language has a “garbage collector” to discover and return unused memory back to the OS. In C++, when using raw pointers for dynamic memory allocation, it is the programmer’s responsibility to free memory when it is no longer needed, returning it to the OS.

The concept of “smart pointers” was added to the language with the C++ 11 standard as part of the standard library, located in the `<memory>` header file. I know `std::auto_ptr` was added with C++ 98, but yuck, and thankfully it is removed with C++ 17. Smart pointers provide a means for automatic memory management of dynamically allocated memory. Generally speaking, a smart pointer manages references to a section of dynamically allocated memory. When the last (smart pointer) reference to the dynamically allocated memory goes out of scope, the smart pointer releases the memory back to the OS. This is not the same as Java’s GC, where the timing of when unused memory is reclaimed is out of the control of the developer. In C++, there developer can trace the code path that leads to when the last reference to the memory goes out of scope, and therefore, when the memory is released; it is deterministic (apart from concurrency).

C++ provides several smart pointers, two of which are discussed in these notes: `std::shared_ptr` (“shared pointer”) and `std::unique_ptr` (“unique pointer). A shared pointer allows a region of dynamically allocated memory to have shared ownership among any number of `std::shared_ptr` objects. A unique pointer enforces a single owner over a region of dynamically allocated memory.

Let’s start with an example, similar to the one used to introduce dynamic memory and raw pointers.

```
std::shared_ptr<int> a = std::make_shared<int>(1);
std::shared_ptr<double> b = std::make_shared<double>(3.14159);

std::cout << "The value stored in a is " << *a << std::endl;
std::cout << "The value stored in b is " << *b << std::endl;
```

Smart pointers are template types. I know we haven’t talked about C++ templates yes, but they are, at a surface level, similar to Java Generics, including the syntax.

The first line declares an `int` shared pointer `a`, followed by allocating a single `int` and initializing it with the value of `1`. Notice the declaration of the shared pointer template type is `int` and not `int*`. When working with smart pointers, the `*` is not used in the type declaration, the `std::shared_ptr` already provides the pointer semantics. Rather than using the `new` operator to make the dynamic memory request, `std::make_shared` is used. The `new` operator returns a raw pointer, but the variable `a` needs an `std::shared_ptr`, which `std::make_shared` returns.

Once a smart pointer variable is defined, the syntax for using it is exactly the same as a raw pointer. This is seen in the two statements that report the values stored inside of `a` and `b` to the console. The dereference operator is used to access the value stored at the dynamically allocated memory, same as a raw pointer.

When the `a` and `b` variables go out of scope, the shared pointers detect this (in their destructors) and release the dynamically allocated memory.

A smart pointer is an object itself, it is a C++ class written as part of the standard library. The first time a region of memory is allocated as a shared pointer a new region of memory is dynamically allocated to fit the request, along with an `std::shared_ptr` instance being made. Internal to the `std::shared_ptr` is a raw pointer that points to the dynamically allocated memory, along with an initial reference count of 1, indicating one `std::shared_ptr` has ownership over the dynamically allocated memory. The reference count is located in another, private, section of dynamically allocated memory. When the `std::shared_ptr` is copied, the reference count is increased by 1. When any (copy or original) `std::shared_ptr` goes out of scope, the reference count is decreased by 1. When the reference count hits 0, it indicates the `std::shared_ptr` that is going out of scope is the last owner of the memory and it then releases the memory back to the OS, along with releasing the memory that holds the reference count.

** Draw a diagram of how a smart (shared) pointer works **

Smart Pointers and Raw Arrays

Let's start by looking at unique pointers and raw arrays. The following code creates a unique pointer to a raw array and demonstrates its use:

```
std::unique_ptr<int[]> primes = std::make_unique<int[]>(4);
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;

for (int index = 0; index < 4; index++)
{
        std::cout << primes[index] << std::endl;
}
```

All looks as you might expect. Unfortunately, an `std::initializer_list` can't be used as the parameter to `std::make_unique`; but we'll overcome that in just a second (resource management), and then revisit again with constructors.

Given the above, it follows for `std::shared_ptr`:

```
std::shared_ptr<int[]> primes = std::make_shared<int[]>(4);
```

VERY surprisingly, this does not compile! It seems *obvious* this is something expected as part of the C++ 11 standard, but for unclear (to me) reasons, it wasn't, but is coming with C++ 20. If you do some reading around the internet, you might find a post from someone who indicates the reason it wasn't included is that no one asked for it!

Until the arrival of C++ 20, it is still desired to have a capability like this, there is a way to do so. This brings us to the interesting capability that smart pointers can be used for manual raw pointer resource management.

Resource Management

An `std::shared_ptr` has an overloaded constructor that accepts a pointer as the first argument and a *deleter* function as the second. A deleter function is one that accepts a pointer to the `std::shared_ptr` type and has a `void` return. The deleter function can do anything, including deleting a raw array! Let's look at an example of how this works:

```
std::shared_ptr<int[]> primes(new int[4]{2, 3, 5, 7}, cleanupArray);
```

Where `cleanupArray` is:

```
void cleanupArray(int* p)
{
    delete[] p;
}
```

The `primes` variable is instantiated using the `std::shared_ptr` overload to accept a pointer to an `int`, or in this case, a raw array, along with the `cleanupArray` deleter function. When the `primes` variable goes out of scope, the `cleanupArray` function is called, which releases the raw array; pretty cool!

(Preventing) The Copy Penalty

There is a small performance penalty each time an `std::shared_ptr` is copied. One of the most common times this occurs is when one is passed as a parameter to a function. By default, C++ is pass-by-value (copy), `std::shared_ptr`s are no different. To avoid the cost associated with making a copy of a parameter, it can be marked as a reference type. However, by declaring the parameter as a reference, it is possible to change the value referred to, which may not be desired in the case of an `std::shared_ptr`. Therefore, the parameter can additionally be marked as `const`. The next function demonstrates this technique.

```
std::uint64_t arraySum(const std::shared_ptr<int[]>& data, std::uint32_t length)
{
    long total = 0;
    for (auto index = 0; index < length; index++)
    {
        total += data[index];
    }

    return total;
}
```

This approach avoids the cost of making a copy of the `std::shared_ptr`, while also preventing an accidental coding error that might change the value of the pointer, while still providing the benefit of using a smart pointer.

While this example shows how to avoid the cost of making a copy of the `std::shared_ptr`, it still allows the data in the array being managed to be changed, which may or may not be desired. We can further strengthen the `const`'ness of the code by declaring the data in the array as `const`, as illustrated next.

```
std::uint64_t arraySum(const std::shared_ptr<const int[]>& data, std::uint32_t length)
{
       ... code goes here ...
}
```

This adds a `const` decorator to the raw array, which tells the compiler to not allow any code that attempts to change the values in the array.

Another approach is to obtain the raw pointer from the `std::shared_ptr` and pass that to a function.

```
std::uint64_t arraySum(const int data[], std::uint32_t length)
{
    long total = 0;
    for (auto index = 0; index < length; index++)
    {
        total += data[index];
    }

    return total;
}
```

This is called as…

```
arraySum(primes.get(), 4);
```

Unique Smart Pointers

Both raw pointers and `std::shared_ptr`'s can be copied over and over without thinking about ownership (too much). Sometimes (think about concurrency) it is desirable to ensure there is one pointer having ownership of an object, and ensure that with a guarantee. Obviously raw pointers can't enforce this, and the semantics of an `std::shared_ptr` is to allow shared ownership of an object. Enter `std::unique_ptr`. An `std::unique_ptr` is very much like an `std::shared_ptr` except that there can only be one `std::unique_ptr` to any object, and this is (sometimes) enforced at compile time. Consider the following code:

```
void reportArray(std::unique_ptr<int[]> data, unsigned int howMany)
{
    for (unsigned int index = 0; index < howMany; index++)
    {
        std::cout << data[index] << std::endl;
    }
}

void demoUniquePointers()
{
    std::unique_ptr<int[]> primes = std::make_unique<int[]>(4);
    primes[0] = 2;
    primes[1] = 3;
    primes[2] = 5;
    primes[3] = 7;

    reportArray(std::move(primes), 4);
}
```

In the function `demoUniquePointers` a (unique) pointer to an array of values is created, then it makes a call to `reportArray` to print out the values. Notice the `std::move` is necessary to "move" the pointer to the function parameter. Without the move operation, the copy constructor is attempted to be used, and it doesn't exist, as a result, a compiler error occurs.

What happens if we add another call to `reportArray` right after the first one? Try it, you'll see the program throws an exception, because the `primes` pointer is no longer valid. The reason primes is no longer valid is the pointer was moved to the function parameter, causing the original pointer to no longer be valid. Then when the function parameter went out of scope, the memory was reclaimed.

We could rewrite the function to take a reference to a unique pointer, that avoids the need to move it and invalidate the original unique pointer. Another alternative is shown next:

```
std::unique_ptr<int[]> reportArray(std::unique_ptr<int[]> data, unsigned int howMany)
{
    for (unsigned int index = 0; index < howMany; index++)
    {
        std::cout << data[index] << std::endl;
    }

    return std::move(primes);
}

void demoUniquePointers()
{
    std::unique_ptr<int[]> primes = std::make_unique<int[]>(4);
    primes[0] = 2;
    primes[1] = 3;
    primes[2] = 5;
    primes[3] = 7;

    primes = reportArray(std::move(primes), 4);
}
```

In this approach, we are being more careful to ensure only one part of the code can access the allocated object, move moving the unique pointer around.

Classes & Dynamic Memory

In Java, all objects (I will use the terms *object* and *instance* interchangeably in these notes) are heap allocated. In C++ an object may be heap or stack allocated, or even partially on the stack and partially heap allocated (depending on the class implementation), it is important to understand how this happens, especially when dealing with pointers.

The following class is used in this section:

```
class Rectangle
{
  public:
    Rectangle(double width, double height) :
        m_width(width),
        m_height(height)
    {
    }

    double getArea() { return m_width * m_height; }
    double getPerimeter() { return m_width * 2 + m_height * 2; }
    double getWidth() { return m_width; }
    double getHeight() { return m_height; }

  private:
    double m_width;
    double m_height;
};
```

This is a simple class used to represent a rectangle and exposes a few methods to obtain details about the object.

Objects – Member Access Syntax

Let's start with the following code:

```
Rectangle r1(2, 4);
std::cout << "(width, height) = ("
    << r1.getWidth() << ", " << r1.getHeight() << ")" << std::endl;
```

The `Rectangle` instance `r1` is stack allocated. While not strictly correct C++ terminology, I will say that `r1` is a value, it is the value of a `Rectangle` object. If you are interested: a value in C++ terminology is something that is *copyable*. It turns out, this `Rectangle` is copyable, but another class implementation may not be copyable.

When working with a value, the `.` (dot) operator is used to access the object members.

Let's move on to the next example:

```
Rectangle* r2 = new Rectangle(4, 6);
std::cout << "(width, height) = ("
    << r2->getWidth() << ", " << r2->getHeight() << ")" << std::endl;
delete r2;
```

The `Rectangle` instance `r2` is heap allocated; using raw pointers. The instance memory is released back to the OS in the `delete r2;` statement.

When working with an object through a pointer, the `->` (pointer operator) is used to access the object members. It doesn't matter if the object is stack or heap allocated, it only matters the type is a pointer. A pointer to a stack allocated object still requires the use of the `->` operator.

Finally, let's look at an example of using smart pointers with a class instance:

```
std::shared_ptr<Rectangle> r3 = std::make_shared<Rectangle>(6, 8);
```

```
        std::cout << "(width, height) = ("
            << r3->getWidth() << ", " << r3->getHeight() << ")" << std::endl;
```

The instance `r3` is heap allocated, using a smart pointer for the memory management. The member access notation for smart pointers is exactly the same as raw pointers, the `->` operator is used for member access.