4. Memory Management

CORE CURRICULUM

Assessment: Memory Management Chatbot due by Aug 4th | 100% VIEWED | PROJECTS

LESSON 1

Introduction

VIEW LESSON →

100% VIEWED SHRINK CARD

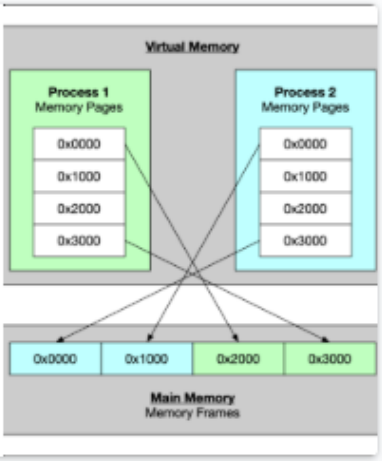
LESSON 2

Overview of Memory Types

This lesson covers basic concepts such as cache, virtual memory, and the structure of memory addresses. In addition, it is demonstrated how the debugger can be used to read data from memory.

VIEW LESSON →

100% VIEWED SHRINK CARD



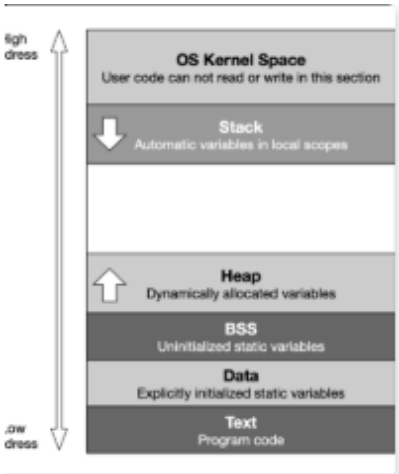
LESSON 3

Variables and Memory

In this lesson the process memory model is introduced, which contains the two fundamental memory areas heap and stack, which play an important role in C++.

VIEW LESSON →

100% VIEWED SHRINK CARD



LESSON 4

Dynamic Memory Allocation (The Heap)

This lesson introduces dynamic memory allocation on the heap. The commands malloc and free as well as new and delete are introduced for this purpose.

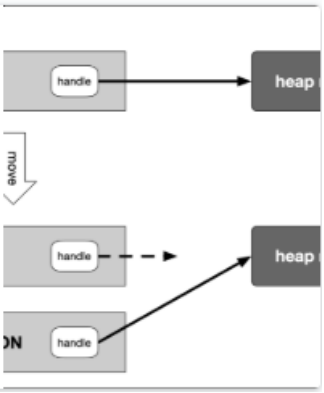
VIEW LESSON →

100% VIEWED SHRINK CARD

LESSON 5

Resource Copying Policies

This section describes how to customize resource copying using the Rule of Three. Also, the Rule of Five is introduced, which helps develop a thorough memory management strategy in your code.



VIEW LESSON →

100% VIEWED SHRINK CARD

LESSON 6

Smart Pointers

In this lesson the three types of smart pointers in C++ are presented and compared. In addition, it is shown how to transfer ownership from one program part to another using copy and move semantics.

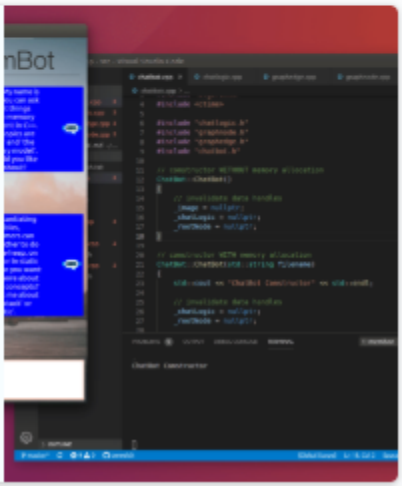
VIEW LESSON →

100% VIEWED SHRINK CARD

PROJECT

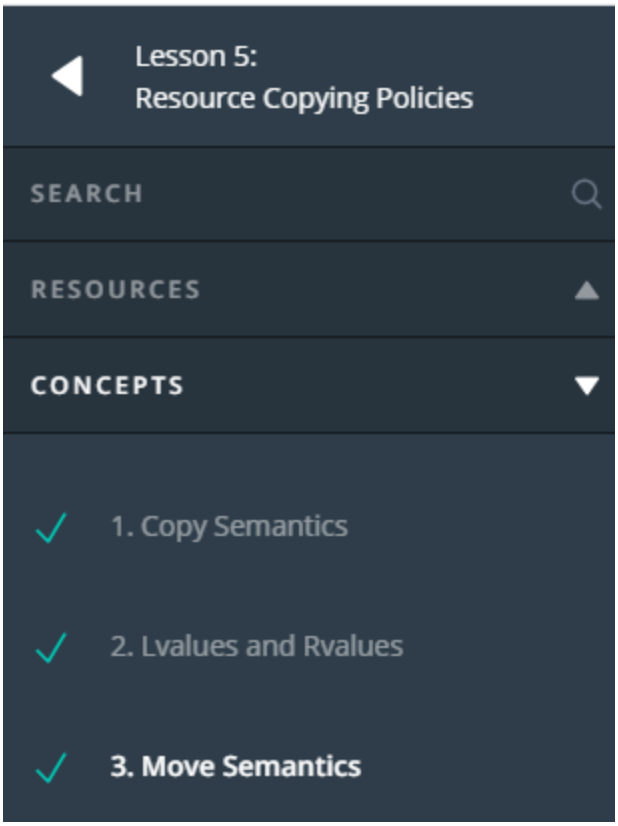
Memory Management Chatbot

The ChatBot project creates a dialogue where users can ask questions about some aspects of memory management in C++. Your task will be to optimize the project with modern memory management in mind.

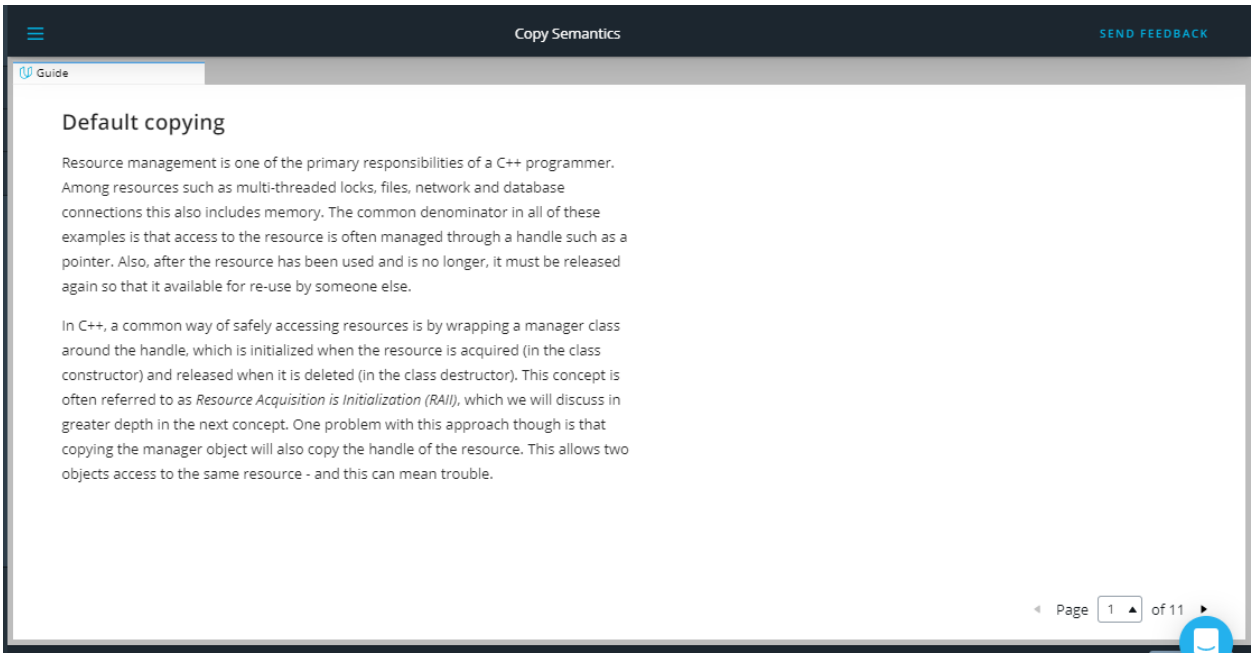


CONTINUE →

STARTED Due in 2 months



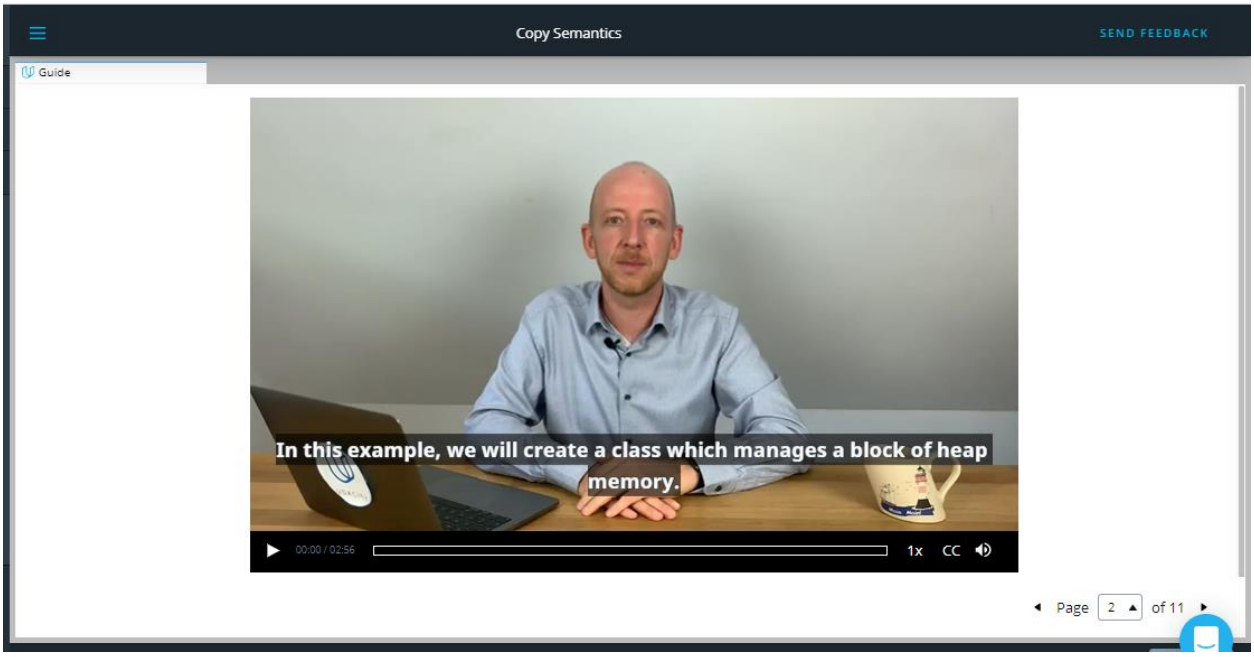
1) Copy Semantics
https://youtu.be/EOzFvKUJt9A



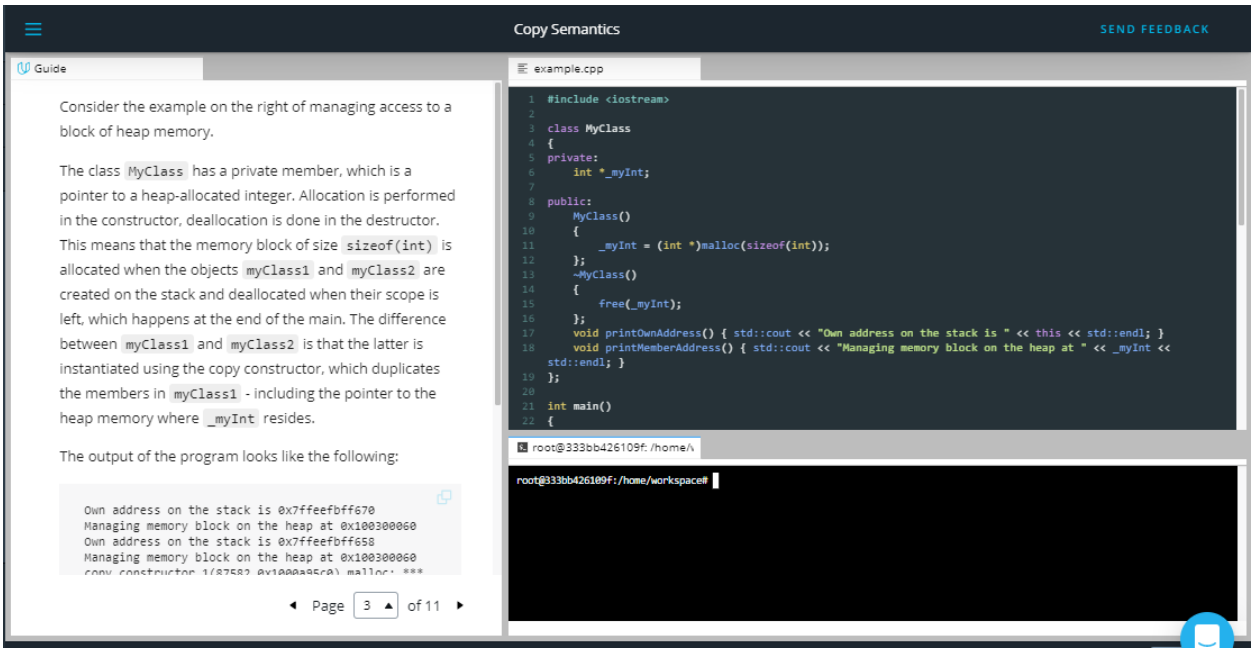
Default copying

Resource management is one of the primary responsibilities of a C++ programmer. Among resources such as multi-threaded locks, files, network and database connections this also includes memory. The common denominator in all of these examples is that access to the resource is often managed through a handle such as a pointer. Also, after the resource has been used and is no longer, it must be released again so that it available for re-use by someone else.

In C++, a common way of safely accessing resources is by wrapping a manager class around the handle, which is initialized when the resource is acquired (in the class constructor) and released when it is deleted (in the class destructor). This concept is often referred to as *Resource Acquisition is Initialization (RAII)*, which we will discuss in greater depth in the next concept. One problem with this approach though is that copying the manager object will also copy the handle of the resource. This allows two objects access to the same resource - and this can mean trouble.



https://video.udacity-data.com/topher/2019/September/5d855b52_nd213-c03-l04-01.2-copy-semantics-sc/nd213-c03-l04-01.2-copy-semantics-sc_720p.mp4

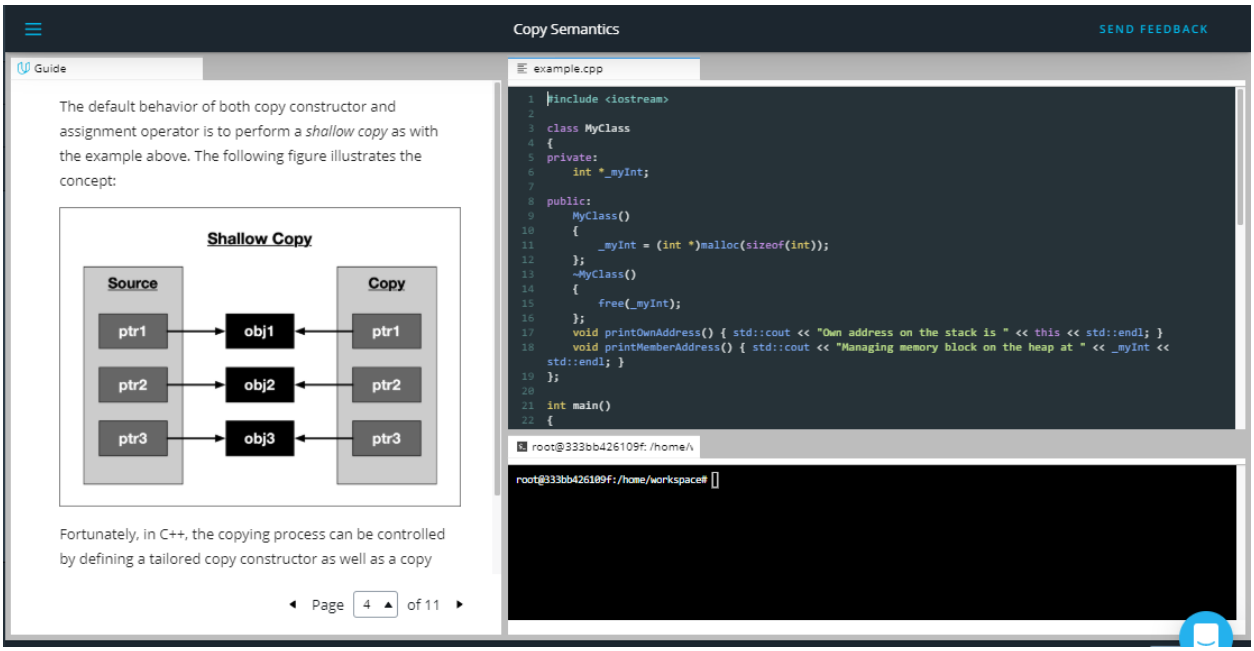


Consider the example on the right of managing access to a block of heap memory.

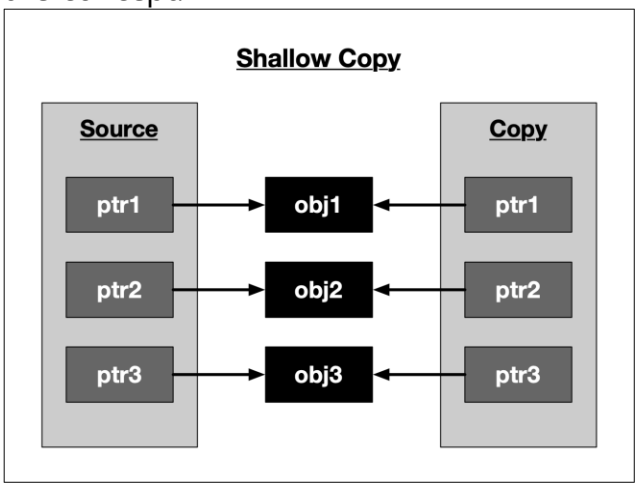
The class MyClass has a private member, which is a pointer to a heap-allocated integer. Allocation is performed in the constructor, deallocation is done in the destructor. This means that the memory block of size sizeof(int) is allocated when the objects myClass1 and myClass2 are created on the stack and deallocated when their scope is left, which happens at the end of the main. The difference between myClass1 and myClass2 is that the latter is instantiated using the copy constructor, which duplicates the members in myClass1 - including the pointer to the heap memory where _myInt resides.
The output of the program looks like the following:

```
Own address on the stack is 0x7ffeefbff670
Managing memory block on the heap at 0x100300060
Own address on the stack is 0x7ffeefbff658
Managing memory block on the heap at 0x100300060
copy_constructor_1(87582,0x1000a95c0) malloc: *** error for object 0x100300060: pointer being freed was not allocated
```

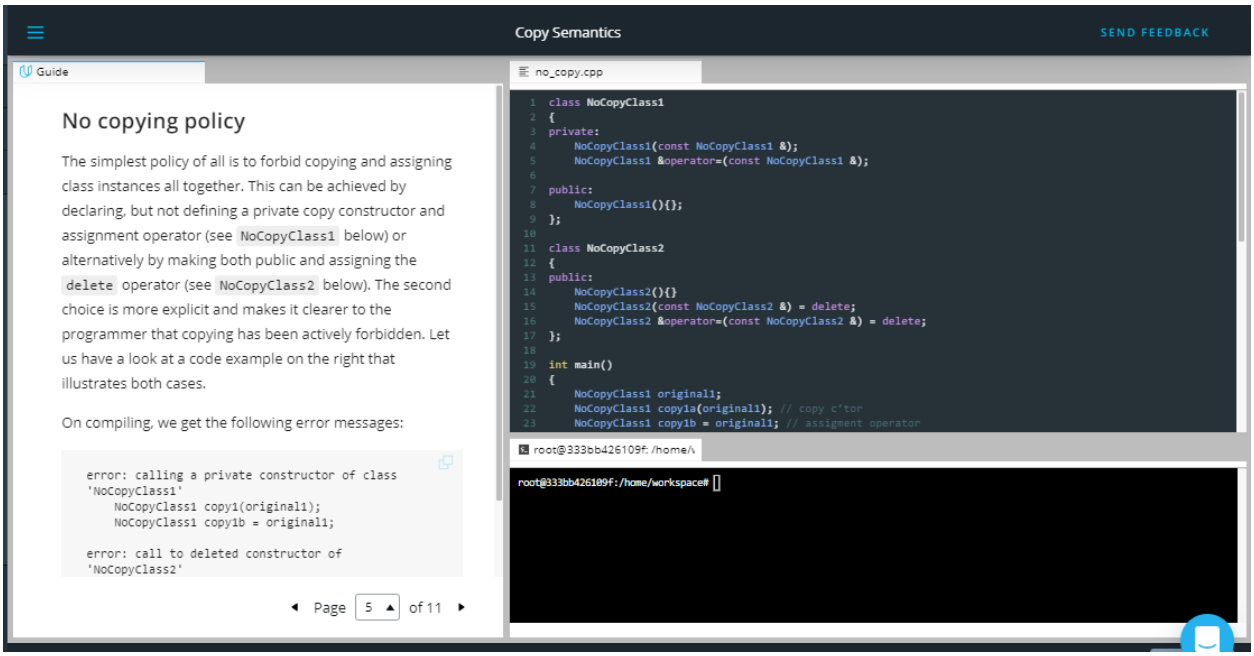
Note that in the workspace, the error will read: *** Error in './a.out': double free or corruption (fasttop): 0x0000000001133c20 ***
From the output we can see that the stack address is different for myClass1 and myClass2 - as was expected. The address of the managed memory block on the heap however is identical. This means that when the first object goes out of scope, it releases the memory resource by calling free in its destructor. The second object does the same - which causes the program to crash as the pointer is now referencing an invalid area of memory, which has already been freed.



The default behavior of both copy constructor and assignment operator is to perform a *shallow copy* as with the example above. The following figure illustrates the concept:



Fortunately, in C++, the copying process can be controlled by defining a tailored copy constructor as well as a copy

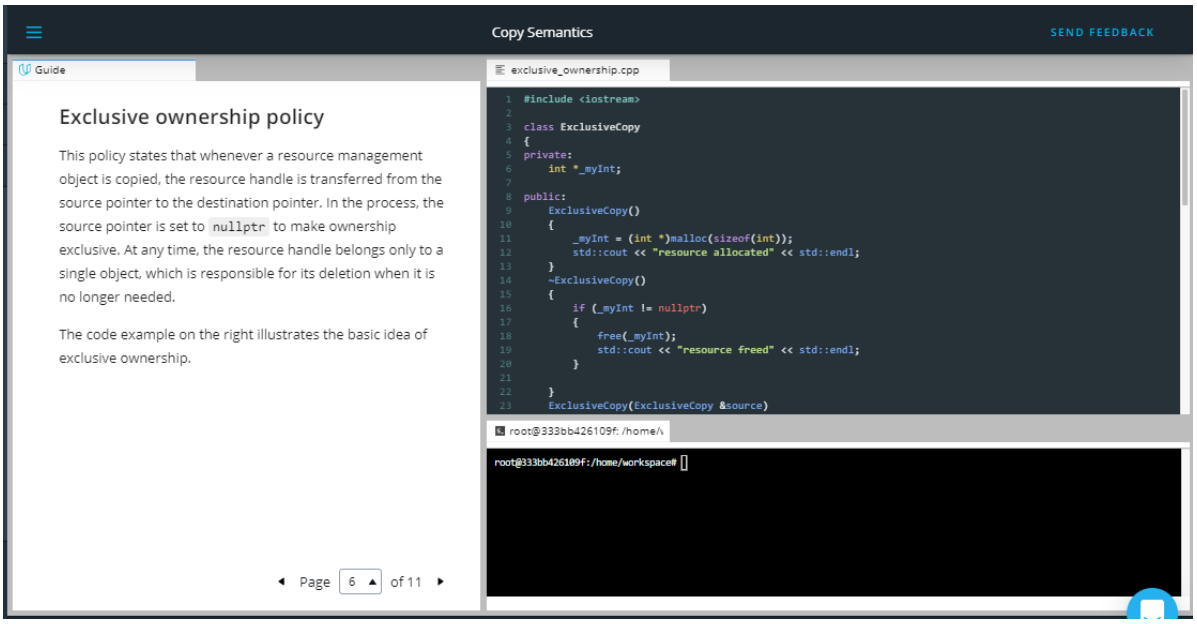


No copying policy

The simplest policy of all is to forbid copying and assigning class instances all together. This can be achieved by declaring, but not defining a private copy constructor and assignment operator (see `NoCopyClass1` below) or alternatively by making both public and assigning the `delete` operator (see `NoCopyClass2` below). The second choice is more explicit and makes it clearer to the programmer that copying has been actively forbidden. Let us have a look at a code example on the right that illustrates both cases.
On compiling, we get the following error messages:

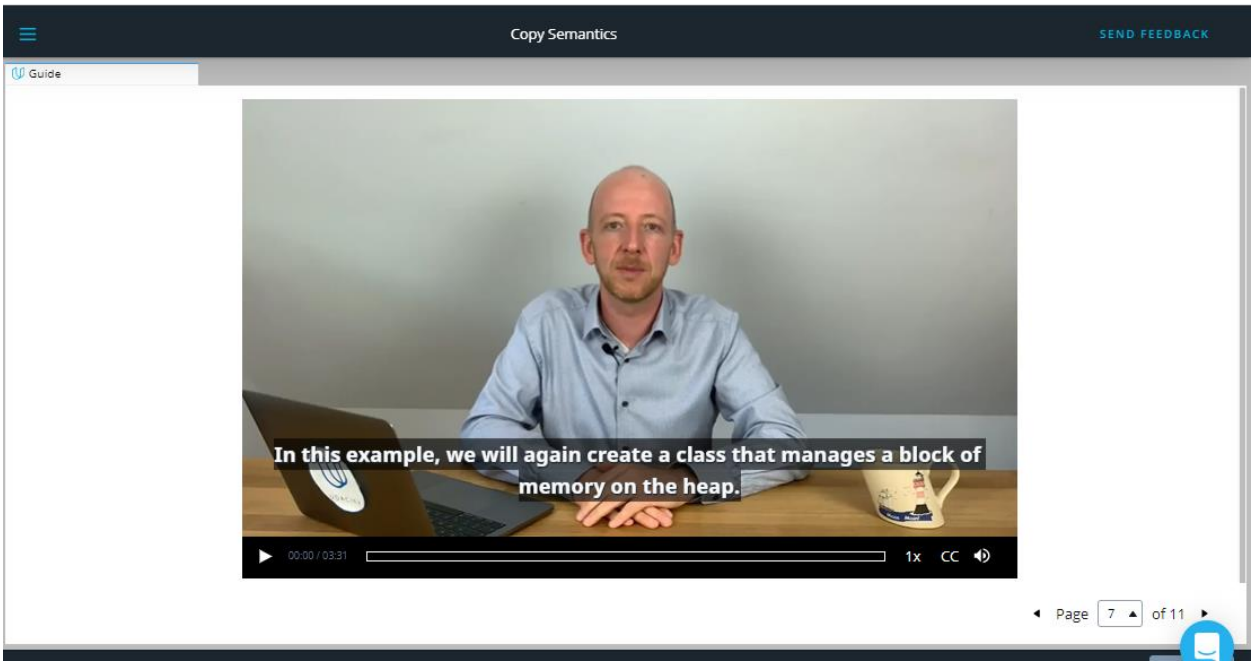
```
error: calling a private constructor of class 'NoCopyClass1'
   NoCopyClass1 copy1(original1);
   NoCopyClass1 copy1b = original1;

error: call to deleted constructor of 'NoCopyClass2'
   NoCopyClass2 copy2(original2);
   NoCopyClass2 copy2b = original2;
```

Both cases effectively prevent the original object from being copied or assigned. In the C++11 standard library, there are some classes for multi-threaded synchronization which use the no copying policy.

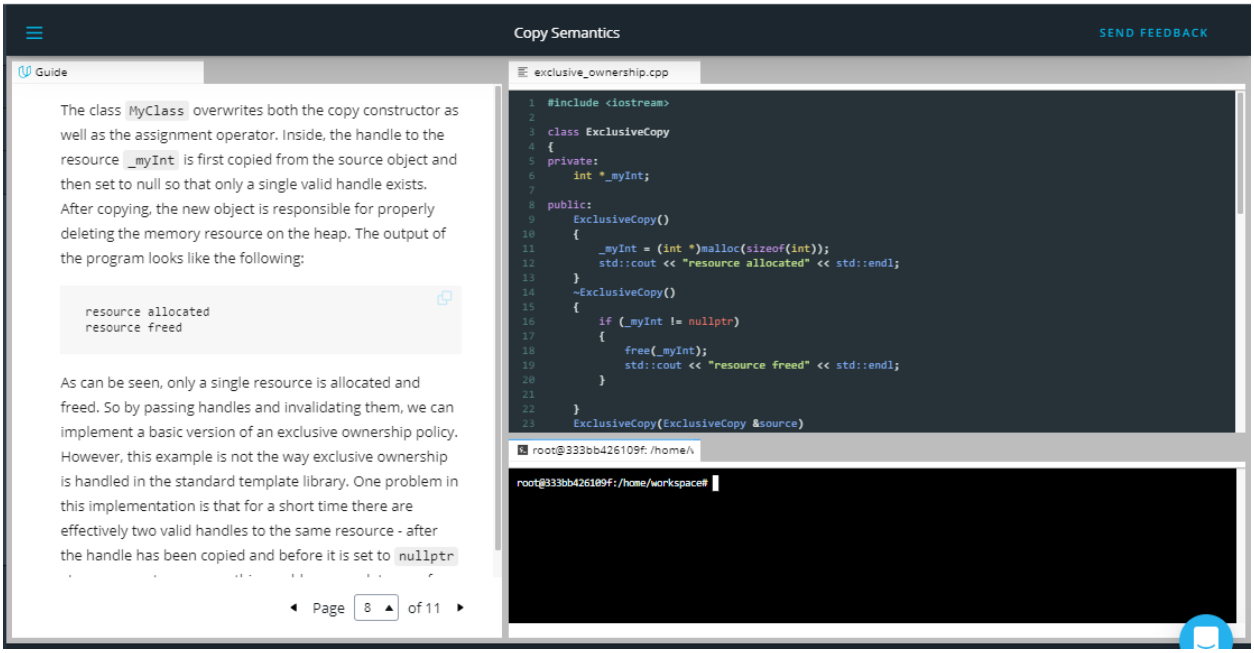


Exclusive ownership policy

This policy states that whenever a resource management object is copied, the resource handle is transferred from the source pointer to the destination pointer. In the process, the source pointer is set to `nullptr` to make ownership exclusive. At any time, the resource handle belongs only to a single object, which is responsible for its deletion when it is no longer needed.
The code example on the right illustrates the basic idea of exclusive ownership.



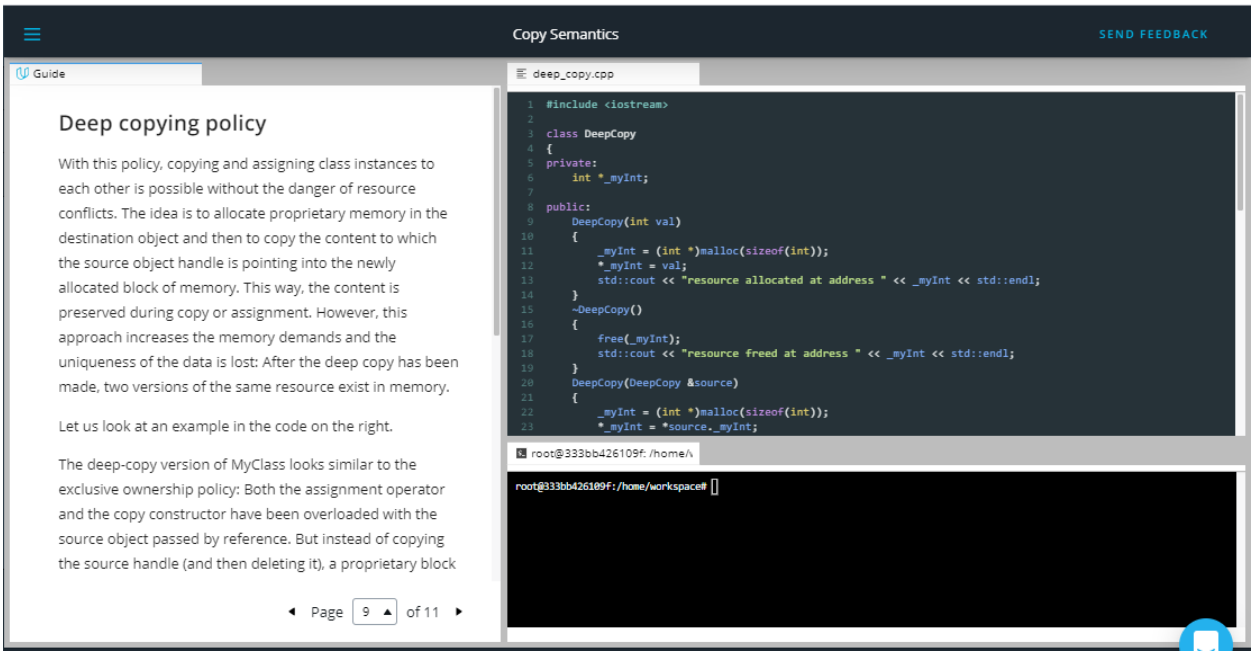
https://video.udacity-data.com/topher/2019/September/5d855b31_nd213-c03-l04-02.2-lvalues-and-rvalues-sc/nd213-c03-l04-02.2-lvalues-and-rvalues-sc_720p.mp4



The class MyClass overwrites both the copy constructor as well as the assignment operator. Inside, the handle to the resource _myInt is first copied from the source object and then set to null so that only a single valid handle exists. After copying, the new object is responsible for properly deleting the memory resource on the heap. The output of the program looks like the following:

```
resource allocated
resource freed
```

As can be seen, only a single resource is allocated and freed. So by passing handles and invalidating them, we can implement a basic version of an exclusive ownership policy. However, this example is not the way exclusive ownership is handled in the standard template library. One problem in this implementation is that for a short time there are effectively two valid handles to the same resource - after the handle has been copied and before it is set to nullptr. In concurrent programs, this would cause a data race for the resource. A much better alternative to handle exclusive ownership in C++ would be to use move semantics, which we will discuss shortly in a very detailed lesson.



Deep copying policy

With this policy, copying and assigning class instances to each other is possible without the danger of resource conflicts. The idea is to allocate proprietary memory in the destination object and then to copy the content to which the source object handle is pointing into the newly allocated block of memory. This way, the content is preserved during copy or assignment. However, this approach increases the memory demands and the uniqueness of the data is lost: After the deep copy has been made, two versions of the same resource exist in memory.

Let us look at an example in the code on the right.

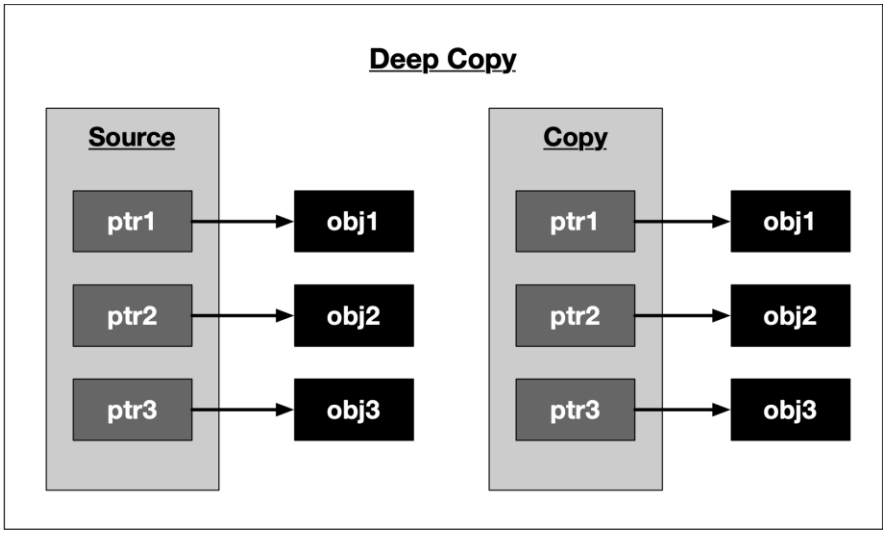
The deep-copy version of MyClass looks similar to the exclusive ownership policy: Both the assignment operator and the copy constructor have been overloaded with the source object passed by reference. But instead of copying the source handle (and then deleting it), a proprietary block of memory is allocated on the heap and the content of the source is copied into it.

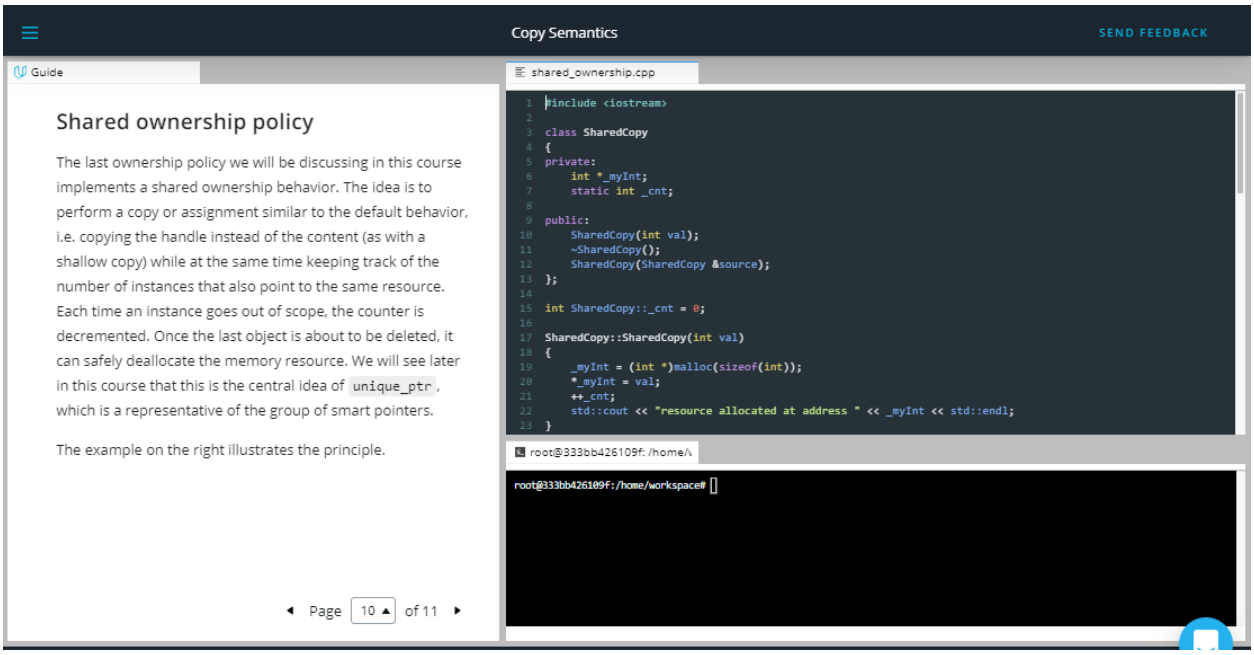
The output of the program looks like the following:

```
resource allocated at address 0x100300060
resource allocated at address 0x100300070 with _myInt = 42
resource allocated at address 0x100300080 with _myInt = 42
resource freed at address 0x100300080
resource freed at address 0x100300070
resource freed at address 0x100300060
```

As can be seen, all copies have the same value of 42 while the address of the handle differs between `source`, `dest1` and `dest2`.

To conclude, the following figure illustrates the idea of a deep copy:

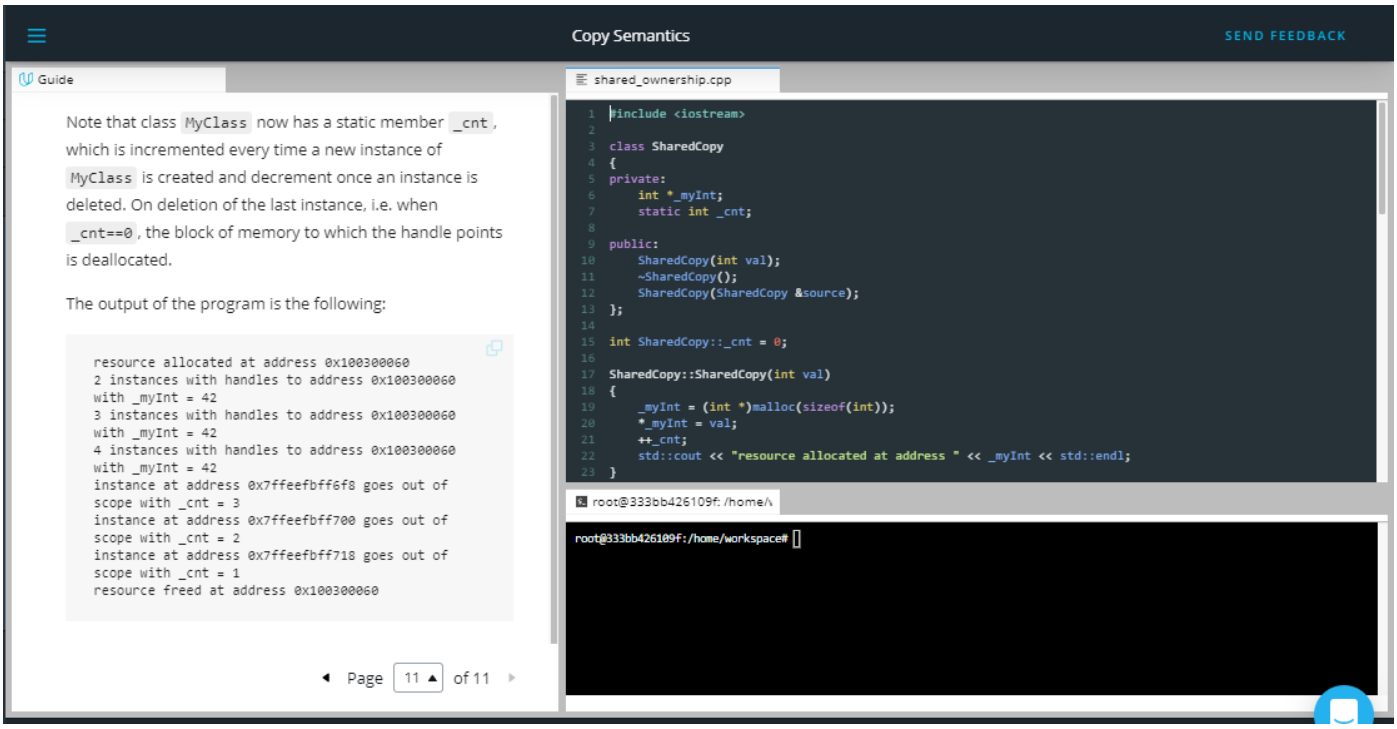




Shared ownership policy

The last ownership policy we will be discussing in this course implements a shared ownership behavior. The idea is to perform a copy or assignment similar to the default behavior, i.e. copying the handle instead of the content (as with a shallow copy) while at the same time keeping track of the number of instances that also

point to the same resource. Each time an instance goes out of scope, the counter is decremented. Once the last object is about to be deleted, it can safely deallocate the memory resource. We will see later in this course that this is the central idea of unique_ptr, which is a representative of the group of smart pointers.
The example on the right illustrates the principle.



Note that class MyClass now has a static member _cnt, which is incremented every time a new instance of MyClass is created and decrement once an instance is deleted. On deletion of the last instance, i.e. when _cnt==0, the block of memory to which the handle points is deallocated.
The output of the program is the following:

```
resource allocated at address 0x100300060
2 instances with handles to address 0x100300060 with _myInt = 42
3 instances with handles to address 0x100300060 with _myInt = 42
4 instances with handles to address 0x100300060 with _myInt = 42
instance at address 0x7ffeefbff6f8 goes out of scope with _cnt = 3
instance at address 0x7ffeefbff700 goes out of scope with _cnt = 2
instance at address 0x7ffeefbff718 goes out of scope with _cnt = 1
resource freed at address 0x100300060
```

As can be seen, the memory is released only once as soon as the reference counter reaches zero.

The Rule of Three

In the previous examples we have taken a first look at several copying policies:

1. Default copying
2. No copying
3. Exclusive ownership
4. Deep copying
5. Shared ownership

In the first example we have seen that the default implementation of the copy constructor does not consider the "special" needs of a class which allocates and deallocates a shared resource on the heap. The problem with implicitly using the default copy constructor or assignment operator is that programmers are not forced to consider the implications for the memory management policy of their program. In the case of the first example, this leads to a segmentation fault and thus a program crash.

In order to properly manage memory allocation, deallocation and copying behavior, we have seen that there is an intricate relationship between destructor, copy constructor and copy assignment operator. To this end, the **Rule of Three** states that if a class needs to have an overloaded copy constructor, copy assignment operator, ~or~ destructor, then it must also implement the other two as well to ensure that memory is managed consistently. As we have seen, the copy constructor and copy assignment operator (which are often almost identical) control how the resource gets copied between objects while the destructor manages the resource deletion.

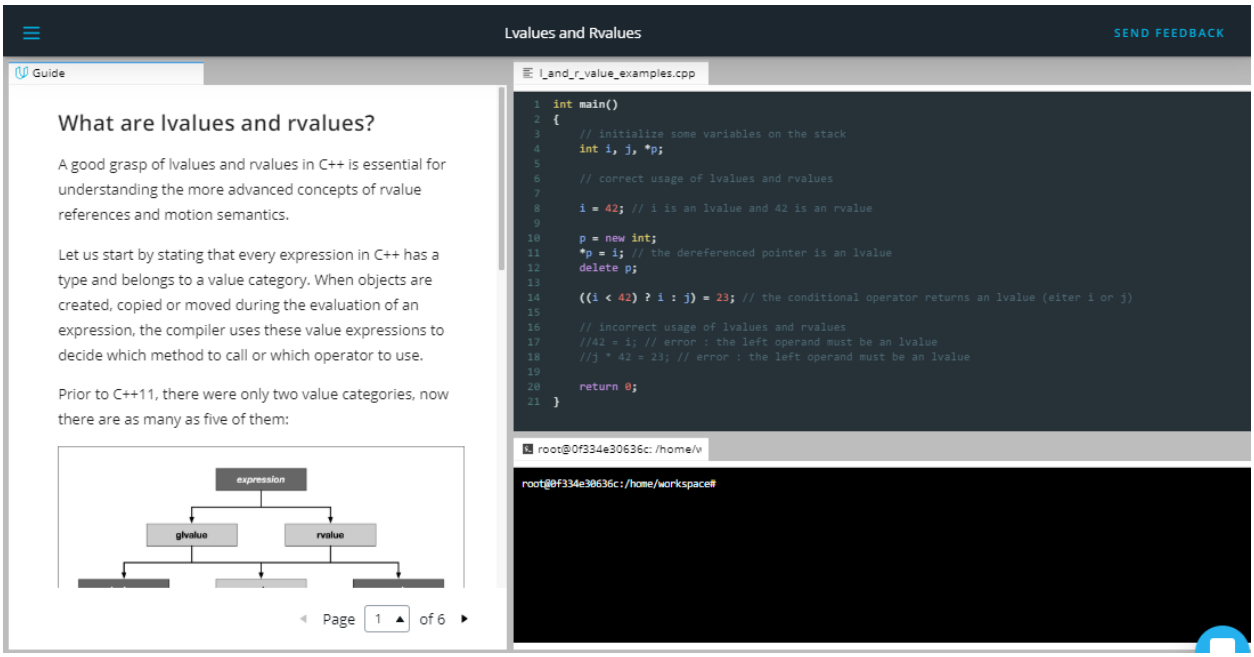
You may have noted that in the previous code example, the class `SharedCopy` does not implement the assignment operator. This is a violation of the **Rule of Three** and thus, if we were to use something like `destination3 = source` instead of `SharedCopy destination3(source)`, the counter variable would not be properly decremented.
The copying policies discussed in this chapter are the basis for a powerful concept in C++11 - smart pointers. But before we discuss these, we need to go into further detail on move semantics, which is a prerequisite you need to learn more about so you can properly understand the exclusive ownership policy as well as the Rule of Five, both of which we will discuss very soon. But before we discuss move semantics, we need to look into the concept of lvalues and rvalues in the next section.

Outro

https://youtu.be/i25M0YmWs7Y

2) Lvalues and Rvalues

https://youtu.be/QCI9DixUUNk

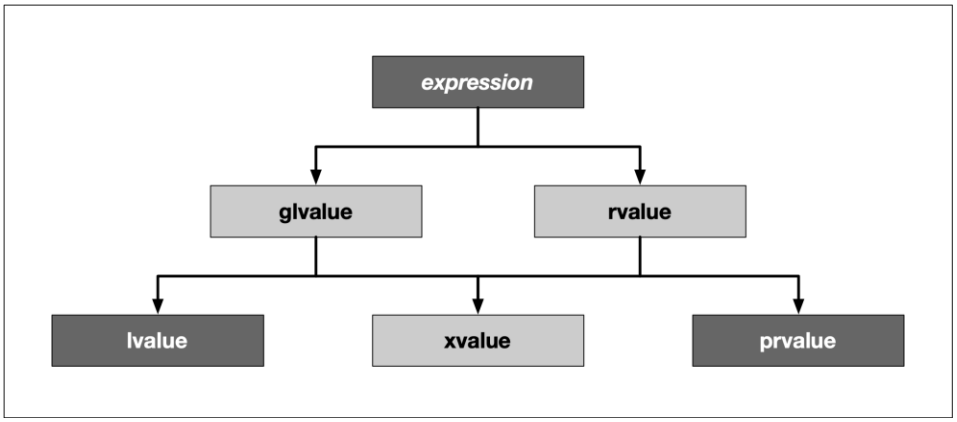


What are lvalues and rvalues?

A good grasp of lvalues and rvalues in C++ is essential for understanding the more advanced concepts of rvalue references and motion semantics.

Let us start by stating that every expression in C++ has a type and belongs to a value category. When objects are created, copied or moved during the evaluation of an expression, the compiler uses these value expressions to decide which method to call or which operator to use.

Prior to C++11, there were only two value categories, now there are as many as five of them:



To keep it short, we do not want to go into all categories, but limit ourselves to lvalues and prvalues:

- **Lvalues** have an address that can be accessed. They are expressions whose evaluation by the compiler determines the identity of objects or functions.
- **Prvalues** do not have an address that is accessible directly. They are temporary expressions used to initialize objects or compute the value of the operand of an operator.

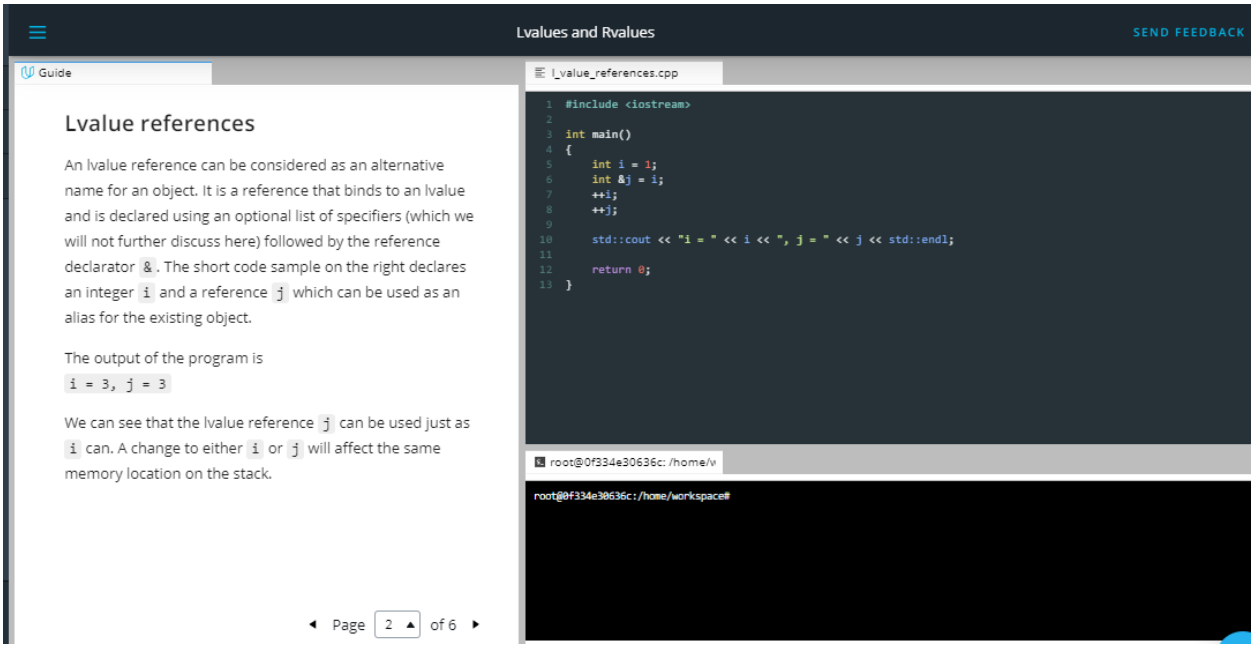
For the sake of simplicity and for compliance with many tutorials, videos and books about the topic, let us refer to *prvalues* as *rvalues* from here on.
The two characters `l` and `r` are originally derived from the perspective of the assignment operator `=`, which always expects a rvalue on the right, and which it assigns to a lvalue on the left. In this case, the `l` stands for left and `r` for right:

```
int i = 42; // lvalue = rvalue;
```

With many other operators, however, this right-left view is not entirely correct. In more general terms, an lvalue is an entity that points to a specific memory location. An rvalue is usually a short-lived object, which is only needed in a narrow local scope. To simplify things a little, one could think of lvalues as *named containers* for rvalues.
In the example above, the value `42` is an rvalue. It does not have a specific memory address which we know about. The rvalue is assigned to a variable `i` with a specific memory location known to us, which is what makes it an lvalue in this example.
Using the address operator `&` we can generate an lvalue from an rvalue and assign it to another lvalue:

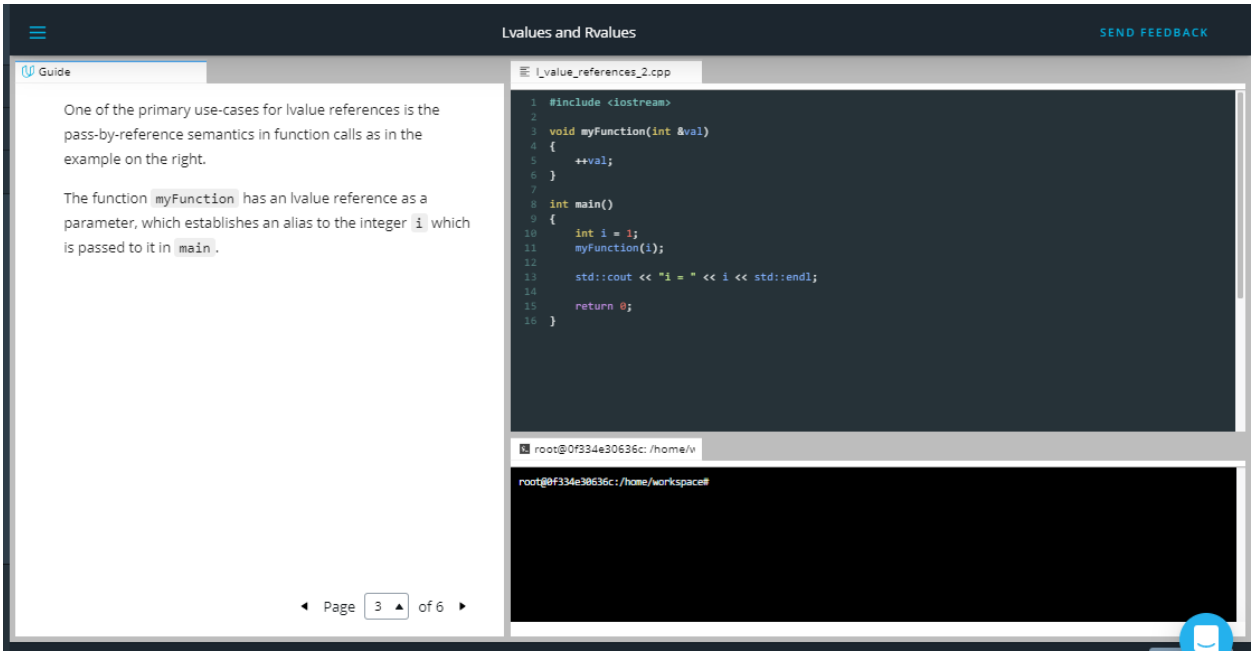
```
int *j = &i;
```

In this small example, the expression `&i` generates the address of `i` as an rvalue and assigns it to `j`, which is an lvalue now holding the memory location of `i`.
The code on the right illustrates several examples of lvalues and rvalues:



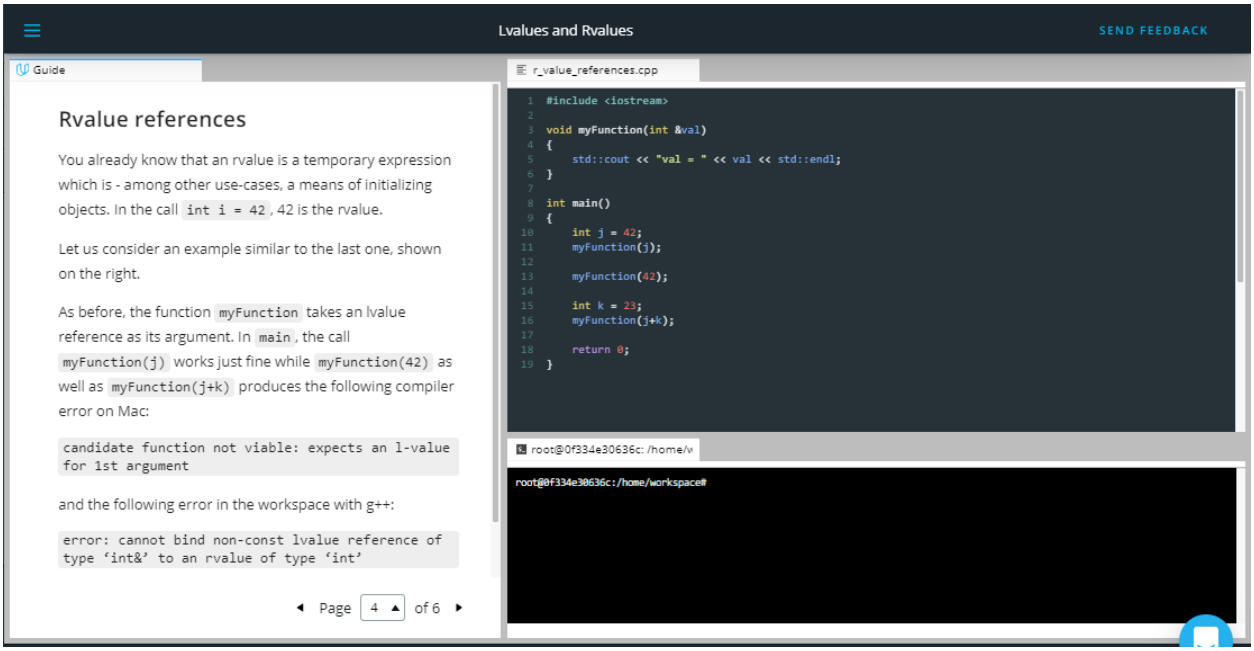
Lvalue references

An lvalue reference can be considered as an alternative name for an object. It is a reference that binds to an lvalue and is declared using an optional list of specifiers (which we will not further discuss here) followed by the reference declarator `&`. The short code sample on the right declares an integer `i` and a reference `j` which can be used as an alias for the existing object.
The output of the program is
`i = 3, j = 3`
We can see that the lvalue reference `j` can be used just as `i` can. A change to either `i` or `j` will affect the same memory location on the stack.



One of the primary use-cases for lvalue references is the pass-by-reference semantics in function calls as in the example on the right.

The function `myFunction` has an lvalue reference as a parameter, which establishes an alias to the integer `i` which is passed to it in `main`.

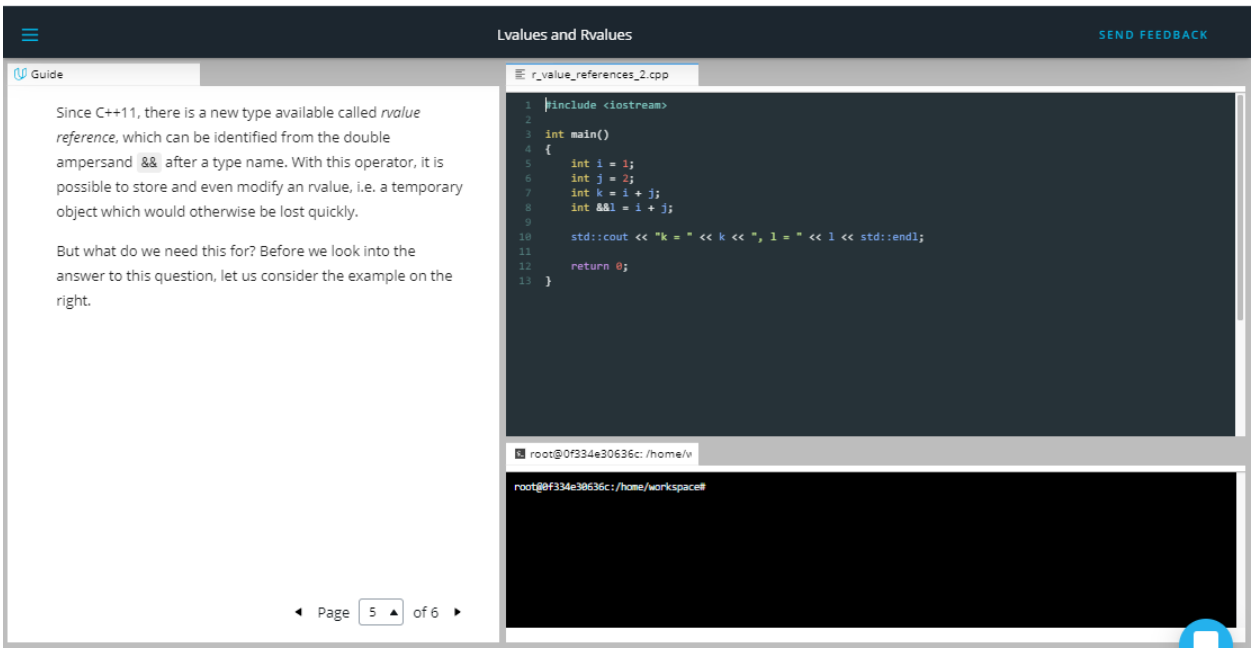


Rvalue references

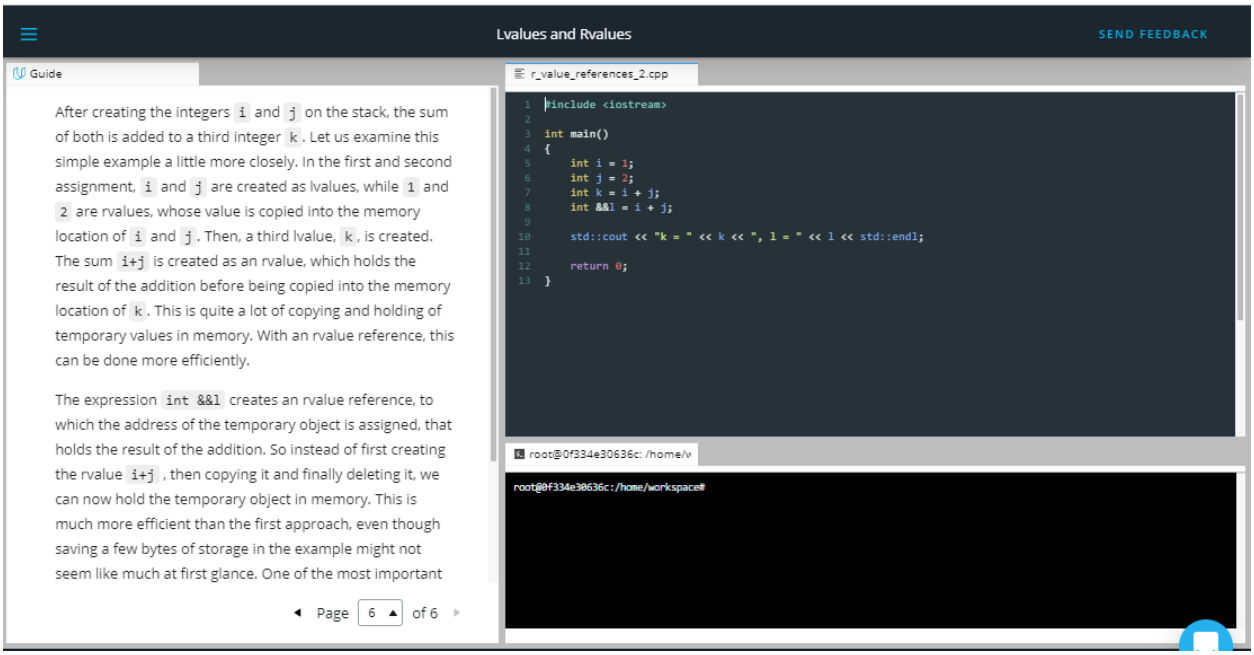
You already know that an rvalue is a temporary expression which is - among other use-cases, a means of initializing objects. In the call `int i = 42`, 42 is the rvalue.
Let us consider an example similar to the last one, shown on the right.

As before, the function `myFunction` takes an lvalue reference as its argument. In `main`, the call `myFunction(j)` works just fine while `myFunction(42)` as well as `myFunction(j+k)` produces the following compiler error on Mac:
`candidate function not viable: expects an l-value for 1st argument`
and the following error in the workspace with g++:

`error: cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'`
While the number `42` is obviously an rvalue, with `j+k` things might not be so obvious, as `j` and `k` are variables and thus lvalues. To compute the result of the addition, the compiler has to create a temporary object to place it in - and this object is an rvalue.



Since C++11, there is a new type available called *rvalue reference*, which can be identified from the double ampersand && after a type name. With this operator, it is possible to store and even modify an rvalue, i.e. a temporary object which would otherwise be lost quickly.
But what do we need this for? Before we look into the answer to this question, let us consider the example on the right.

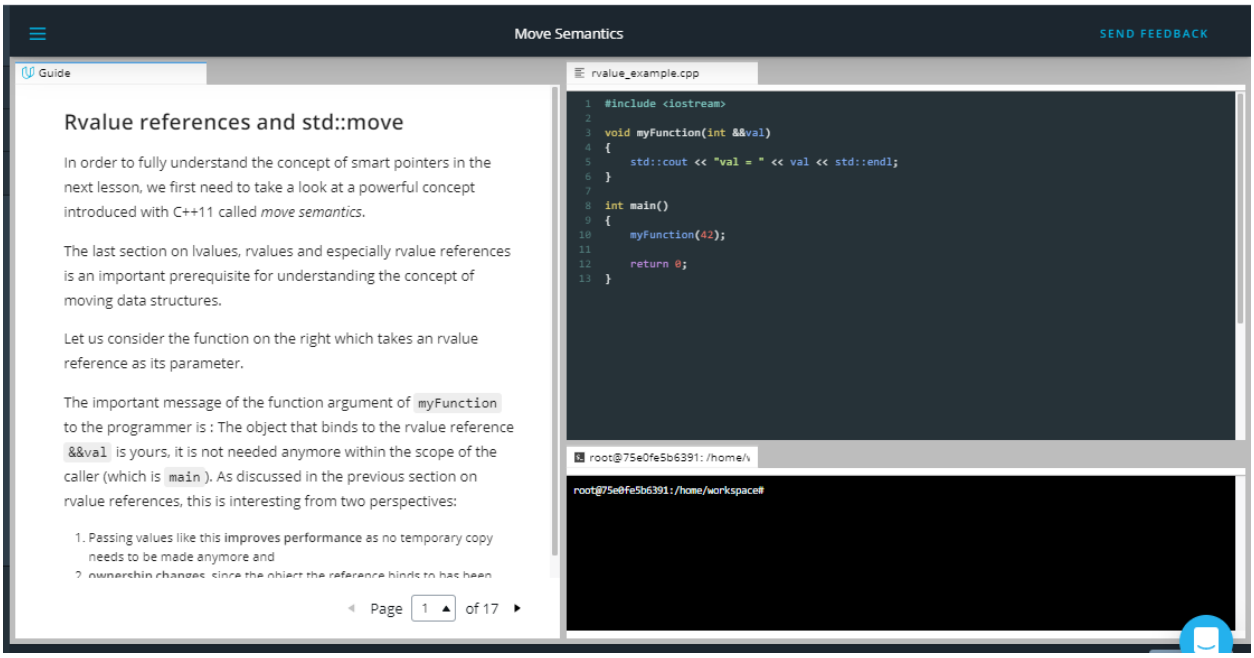


After creating the integers `i` and `j` on the stack, the sum of both is added to a third integer `k`. Let us examine this simple example a little more closely. In the first and second assignment, `i` and `j` are created as lvalues, while `1` and `2` are rvalues, whose value is copied into the memory location of `i` and `j`. Then, a third lvalue, `k`, is created. The sum `i+j` is created as an rvalue, which holds the result of the addition before being copied into the memory location of `k`. This is quite a lot of copying and holding of temporary values in memory. With an rvalue reference, this can be done more efficiently.
The expression `int &&l` creates an rvalue reference, to which the address of the temporary object is assigned, that holds the result of the addition. So instead of first creating the rvalue `i+j`, then copying it and finally deleting it, we can now hold the temporary object in memory. This is much more efficient than the first approach, even though saving a few bytes of storage in the example might not seem like much at first glance. One of the most important aspects of rvalue references is that they pave the way for *move semantics*, which is a mighty technique in modern C++ to optimize memory usage and processing speed. Move semantics and rvalue references make it possible to write code that transfers resources such as dynamically allocated memory from one object to another in a very efficient manner and also supports the concept of exclusive ownership, as we will shortly see when discussing smart pointers. In the next section we will take a close look at move semantics and its benefits for memory management.

Outro

https://youtu.be/3cKhShD1lD4

3) Move Semantics
   https://youtu.be/s6IJqVOQN0A



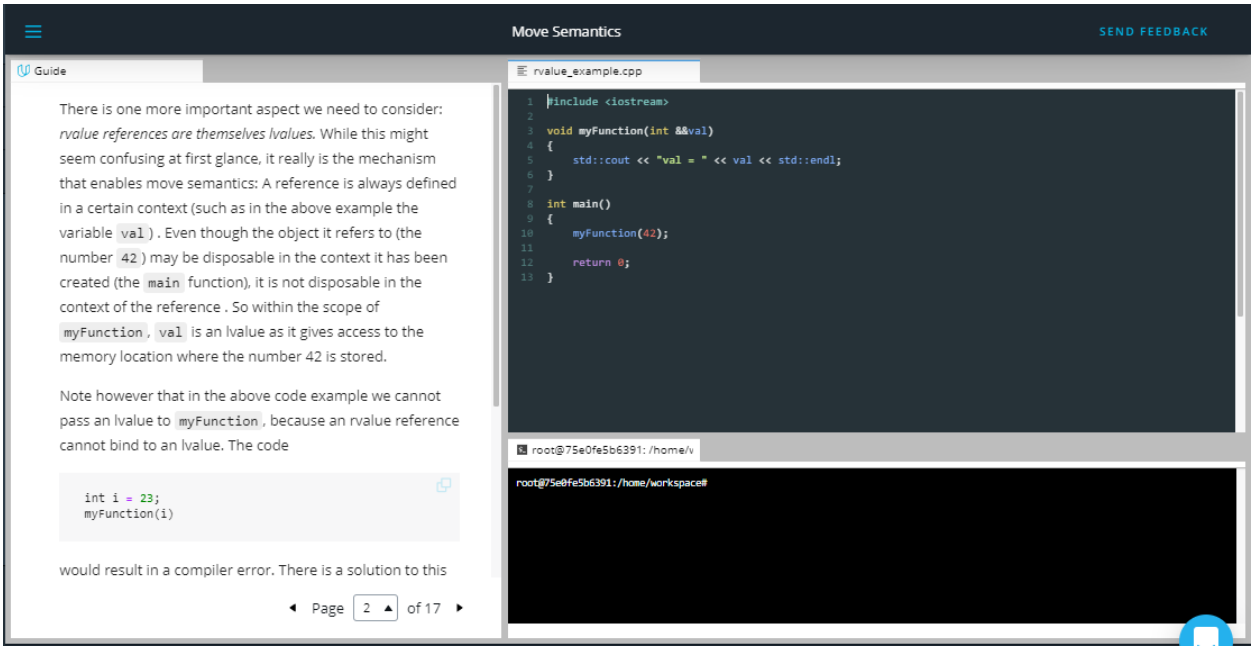
Rvalue references and std::move

In order to fully understand the concept of smart pointers in the next lesson, we first need to take a look at a powerful concept introduced with C++11 called *move semantics*.
The last section on lvalues, rvalues and especially rvalue references is an important prerequisite for understanding the concept of moving data structures.

Let us consider the function on the right which takes an rvalue reference as its parameter.

The important message of the function argument of `myFunction` to the programmer is : The object that binds to the rvalue reference `&&val` is yours, it is not needed anymore within the scope of the caller (which is `main`). As discussed in the previous section on rvalue references, this is interesting from two perspectives:

1. Passing values like this **improves performance** as no temporary copy needs to be made anymore and
2. **ownership changes**, since the object the reference binds to has been abandoned by the caller and now binds to a handle which is available only to the receiver. This could not have been achieved with lvalue references as any change to the object that binds to the lvalue reference would also be visible on the caller side.



There is one more important aspect we need to consider: *rvalue references are themselves lvalues.* While this might seem confusing at first glance, it really is the mechanism that enables move semantics: A reference is always defined in a certain context (such as in the above example the variable `val`). Even though the object it refers to (the number `42`) may be disposable in the context it has been created (the `main` function), it is not disposable in the context of the reference. So within the scope of `myFunction`, `val` is an lvalue as it gives access to the memory location where the number 42 is stored.

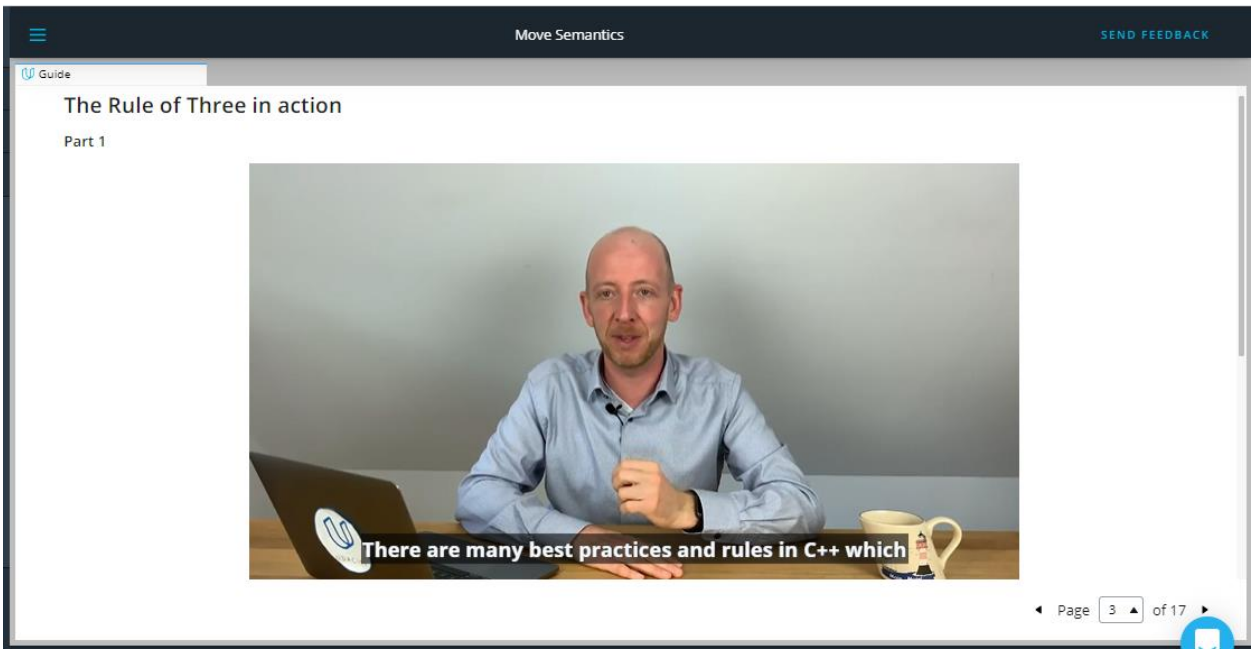
Note however that in the above code example we cannot pass an lvalue to `myFunction`, because an rvalue reference cannot bind to an lvalue. The code

```
int i = 23;
myFunction(i)
```

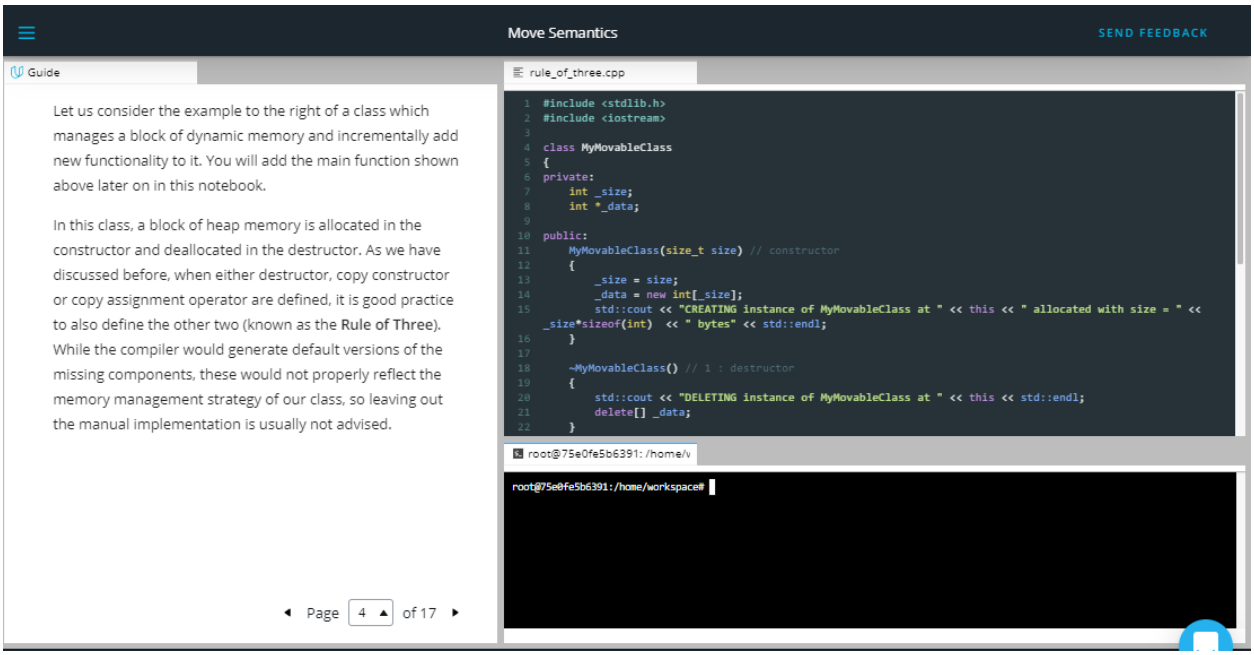
would result in a compiler error. There is a solution to this problem though: The function `std::move` converts an lvalue into an rvalue (actually, to be exact, into an *xvalue*, which we will not discuss here for the sake of clarity), which makes it possible to use the lvalue as an argument for the function:

```
int i = 23;
myFunction(std::move(i));
```

In doing this, we state that in the scope of `main` we will not use i anymore, which now exists only in the scope of `myFunction`. Using `std::move` in this way is one of the components of move semantics, which we will look into shortly. But first let us consider an example of the **Rule of Three**.



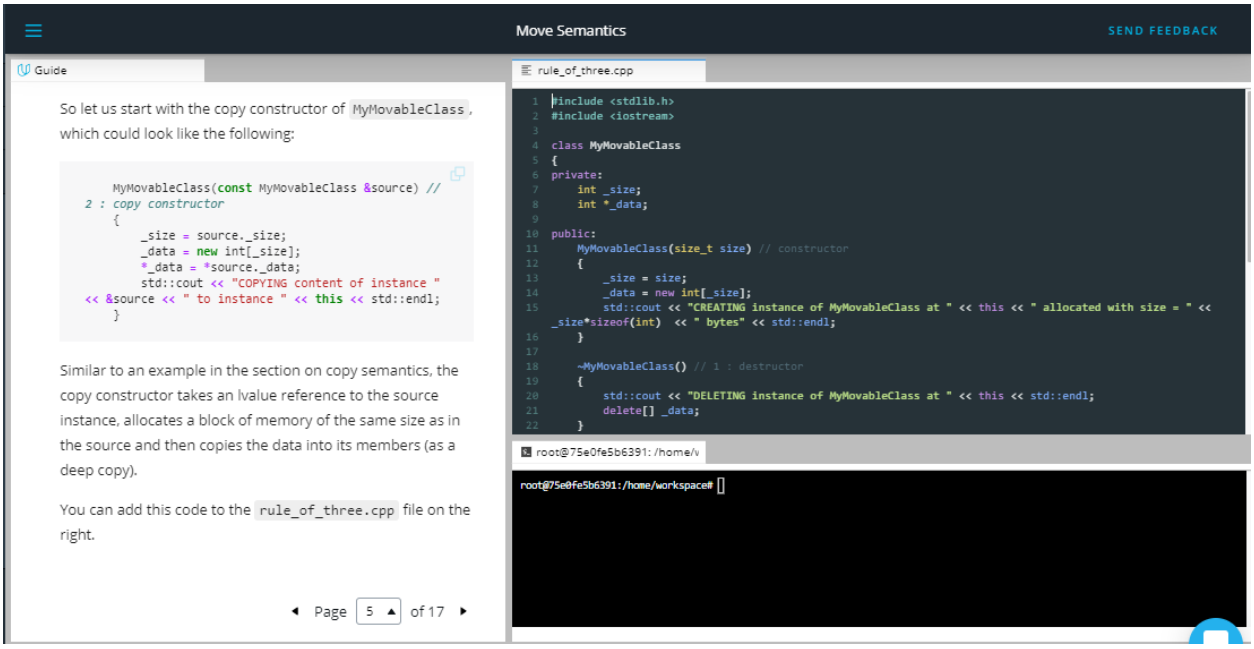
https://video.udacity-data.com/topher/2019/September/5d855b0c_nd213-c03-l04-03.2-move-semantics-sc/nd213-c03-l04-03.2-move-semantics-sc_720p.mp4



Let us consider the example to the right of a class which manages a block of dynamic memory and incrementally add new functionality to it. You will add the main function shown above later on in this notebook.

In this class, a block of heap memory is allocated in the constructor and deallocated in the destructor. As we have discussed before, when either destructor, copy constructor or copy assignment operator are defined, it is good practice to also

define the other two (known as the **Rule of Three**). While the compiler would generate default versions of the missing components, these would not properly reflect the memory management strategy of our class, so leaving out the manual implementation is usually not advised.

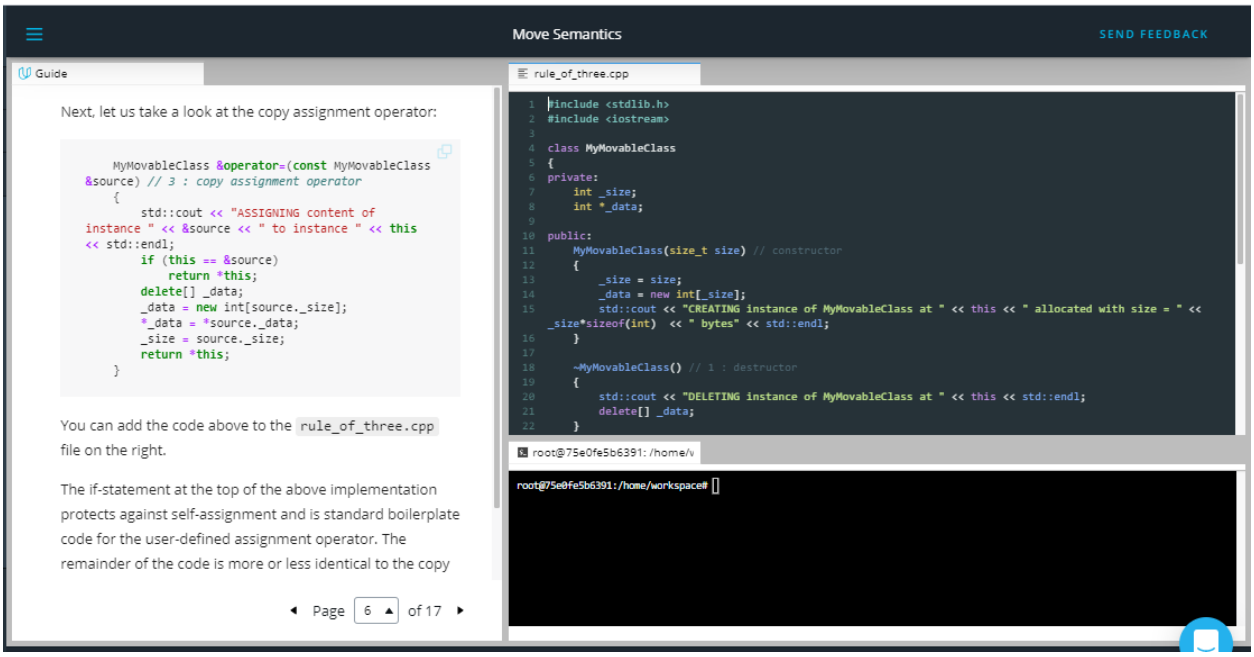


So let us start with the copy constructor of MyMovableClass, which could look like the following:

```
MyMovableClass(const MyMovableClass &source) //2 : copy constructor
{
  _size = source._size;
  _data = new int[_size];
  *_data = *source._data;
  std::cout << "COPYING content of instance " << &source << " to instance " << this << std::endl;
}
```

Similar to an example in the section on copy semantics, the copy constructor takes an lvalue reference to the source instance, allocates a block of memory of the same size as in the source and then copies the data into its members (as a deep copy).

You can add this code to the rule_of_three.cpp file on the right.



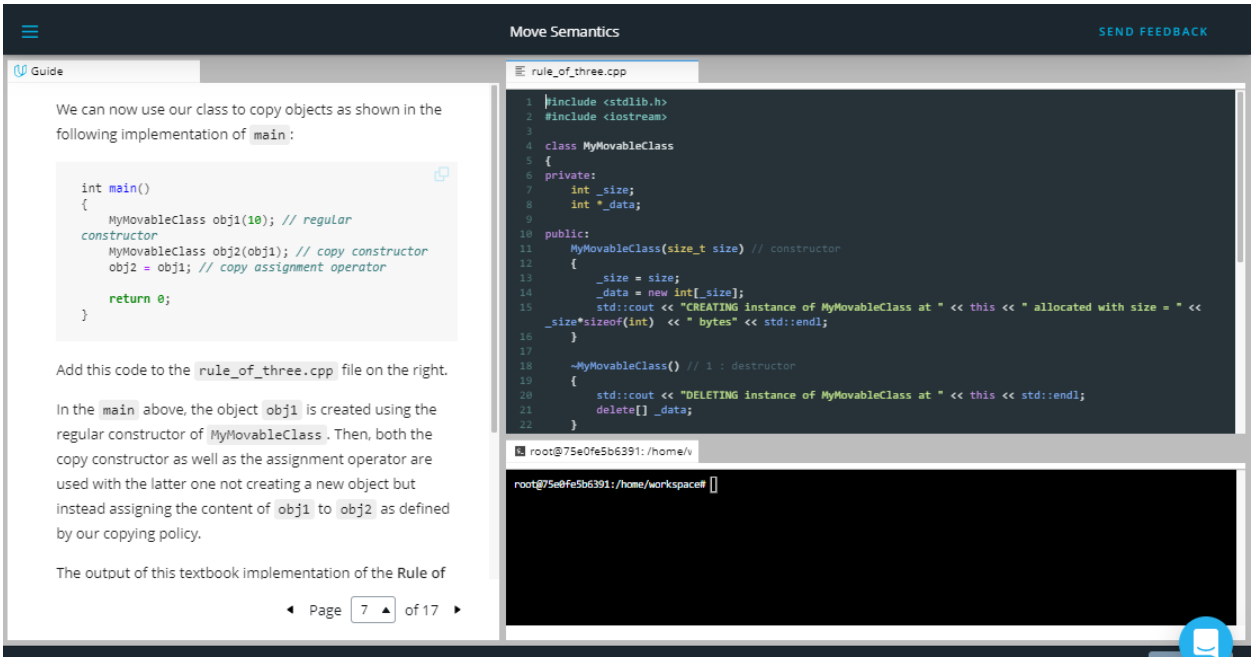
Next, let us take a look at the copy assignment operator:

```
MyMovableClass &operator=(const MyMovableClass &source) // 3 : copy assignment operator
{
  std::cout << "ASSIGNING content of instance " << &source << " to instance " << this << std::endl;
  if (this == &source)
    return *this;
  delete[] _data;
  _data = new int[source._size];
  *_data = *source._data;
  _size = source._size;
  return *this;
}
```

You can add the code above to the rule_of_three.cpp file on the right.

The if-statement at the top of the above implementation protects against self-assignment and is standard boilerplate code for the user-defined assignment operator. The remainder of the code is more or less identical to the copy constructor, apart from returning a reference to the own instance using this.

You might have noticed that both copy constructor and assignment operator take a const reference to the source object as an argument, by which they promise that they won' (and can't) modify the content of source.



We can now use our class to copy objects as shown in the following implementation of main:

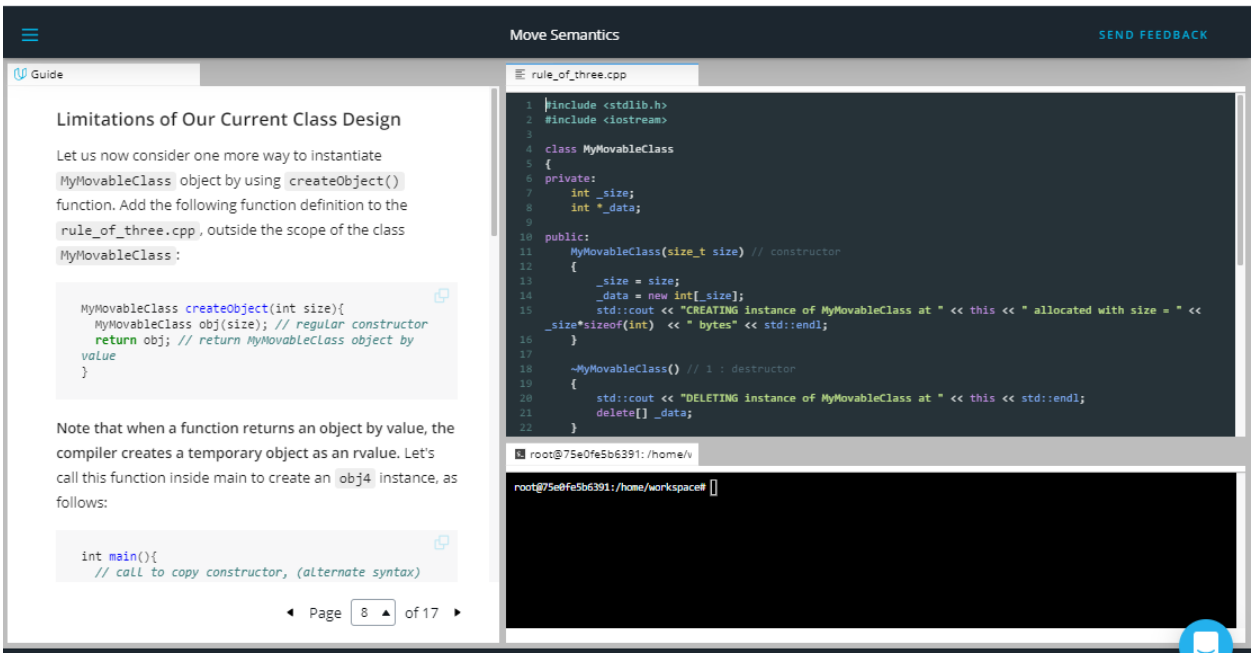
```
int main()
{
   MyMovableClass obj1(10); // regular constructor
   MyMovableClass obj2(obj1); // copy constructor
   obj2 = obj1; // copy assignment operator

   return 0;
}
```

Add this code to the rule_of_three.cpp file on the right.

In the main above, the object obj1 is created using the regular constructor of MyMovableClass. Then, both the copy constructor as well as the assignment operator are used with the latter one not creating a new object but instead assigning the content of obj1 to obj2 as defined by our copying policy.

The output of this textbook implementation of the **Rule of Three** looks like this:

```
CREATING instance of MyMovableClass at 0x7ffeefbff618 allocated with size = 40 bytes

COPYING content of instance 0x7ffeefbff618 to instance 0x7ffeefbff608

ASSIGNING content of instance 0x7ffeefbff618 to instance 0x7ffeefbff608

DELETING instance of MyMovableClass at 0x7ffeefbff608

DELETING instance of MyMovableClass at 0x7ffeefbff618
```



Limitations of Our Current Class Design

Let us now consider one more way to instantiate `MyMovableClass` object by using `createObject()` function. Add the following function definition to the `rule_of_three.cpp`, outside the scope of the class `MyMovableClass`:

```
MyMovableClass createObject(int size){
  MyMovableClass obj(size); // regular constructor
  return obj; // return MyMovableClass object by value
}
```

Note that when a function returns an object by value, the compiler creates a temporary object as an rvalue. Let's call this function inside main to create an `obj4` instance, as follows:

```
int main(){
  // call to copy constructor, (alternate syntax)
  MyMovableClass obj3 = obj1;
  // Here, we are instantiating obj3 in the same statement; hence the copy assignment operator would not be called.

  MyMovableClass obj4 = createObject(10);
  // createObject(10) returns a temporary copy of the object as an rvalue, which is passed to the copy constructor.


  /*
  * You can try executing the statement below as well
  * MyMovableClass obj4(createObject(10));
  */

  return 0;
}
```

In the main above, the returned value of createObject(10) is passed to the copy constructor. The function createObject() returns an instance of MyMovableClass by value. In such a case, the compiler creates a temporary copy of the object as an rvalue, which is passed to the copy constructor.

A special call to copy constructor

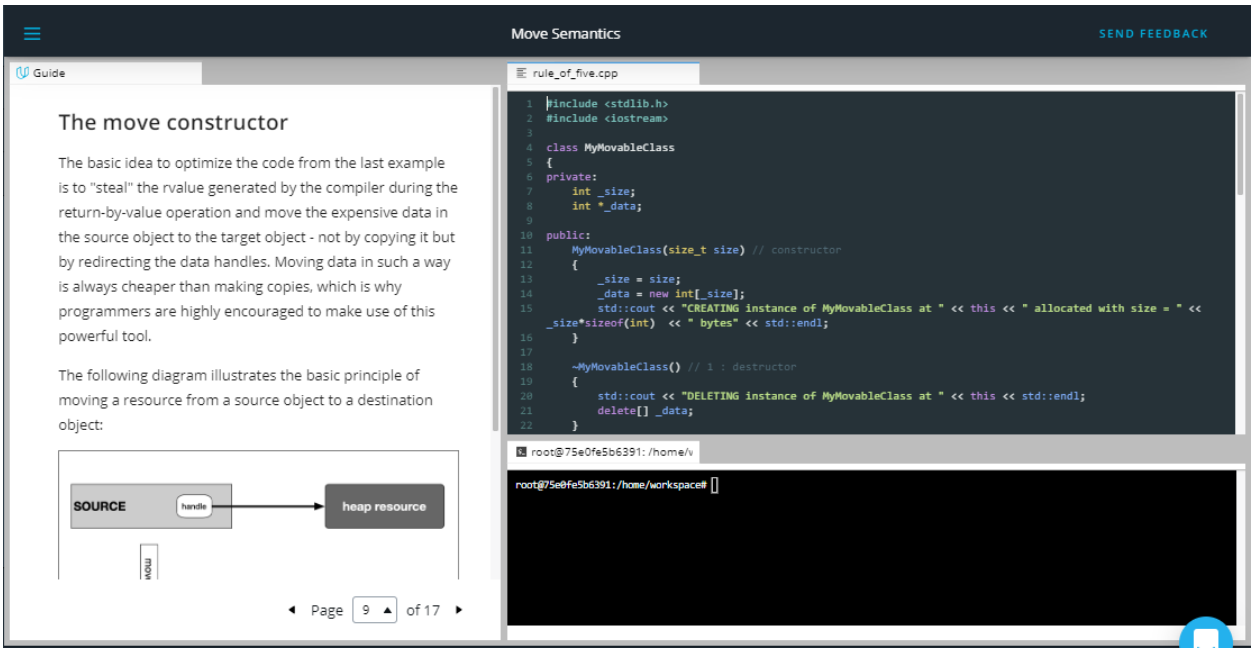
Try compiling and then running the rule_of_three.cpp to notice that MyMovableClass obj4 = createObject(10); would not print the cout statement of copy constructor on the console. This is because the copy constructor is called on the temporary object.

In our current class design, while creating obj4, the data is dynamically allocated on the stack, which is then copied from the temporary object to its target destination. This means that **two expensive memory operations** are performed with the first occurring during the creation of the temporary rvalue and the second during the execution of the copy constructor. The similar two expensive memory operations would be performed with the assignment operator if we execute the following statement inside main:

```
MyMovableClass obj4 = createObject(10); // Don't write this statement if you have already written it before
obj4 = createObject(10); // call to copy assignment operator
```

In the above call to copy assignment operator, it would first erase the memory of obj4, then reallocate it during the creation of the temporary object; and then copy the data from the temporary object to obj4.

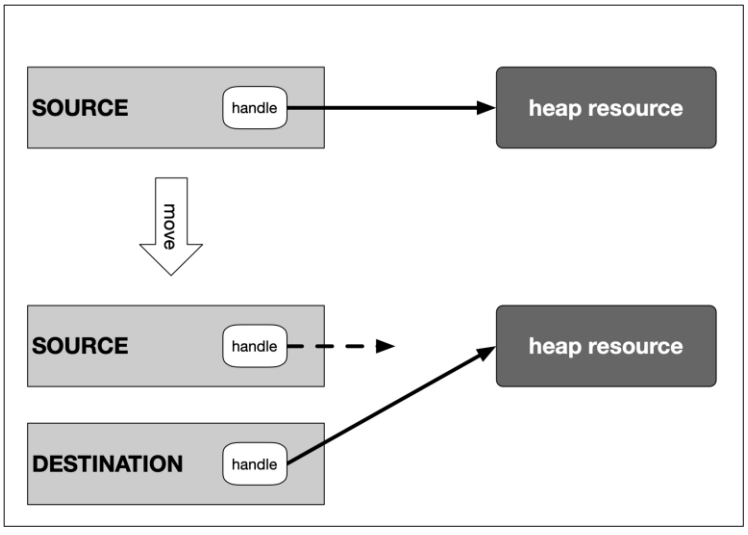
From a performance viewpoint, this code involves far too many copies, making it inefficient - especially with large data structures. Prior to C++11, the proper solution in such a case was to simply avoid returning large data structures by value to prevent the expensive and unnecessary copying process. With C++11 however, there is a way we can optimize this and return even large data structures by value. The solution is the move constructor and the **Rule of Five**.



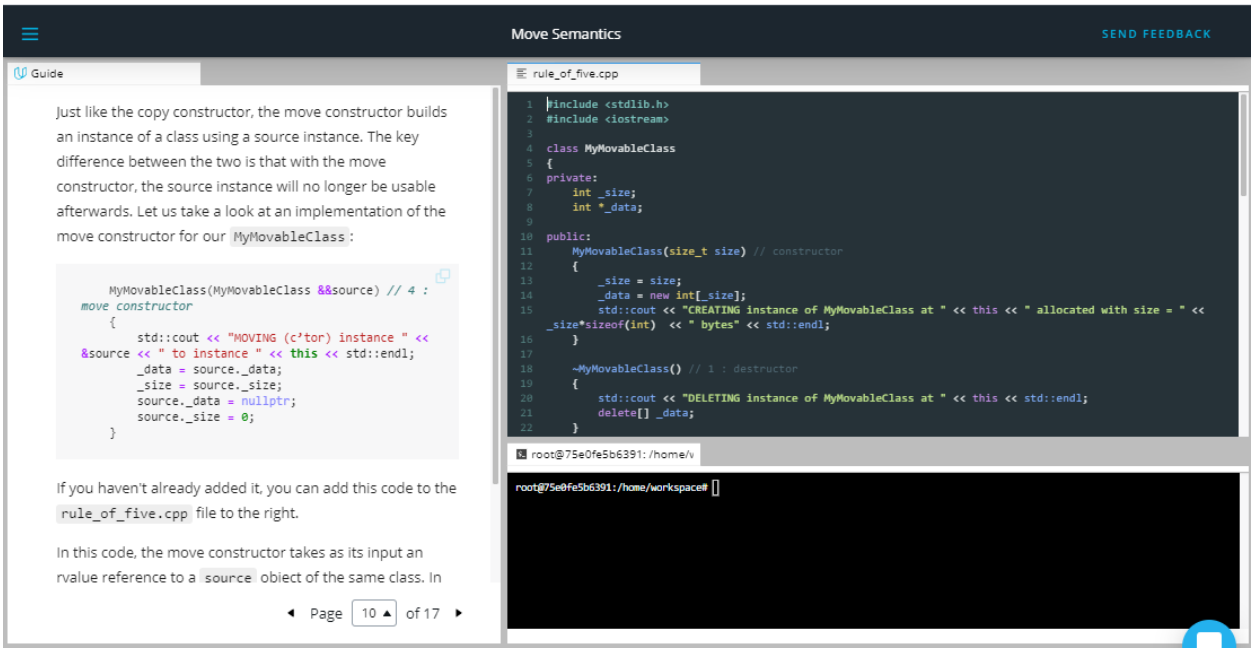
The move constructor

The basic idea to optimize the code from the last example is to "steal" the rvalue generated by the compiler during the return-by-value operation and move the expensive data in the source object to the target object - not by copying it but by redirecting the data handles. Moving data in such a way is always cheaper than making copies, which is why programmers are highly encouraged to make use of this powerful tool.

The following diagram illustrates the basic principle of moving a resource from a source object to a destination object:



In order to achieve this, we will be using a construct called *move constructor*, which is similar to the copy constructor with the key difference being the re-use of existing data without unnecessarily copying it. In addition to the move constructor, there is also a move assignment operator, which we need to look at.



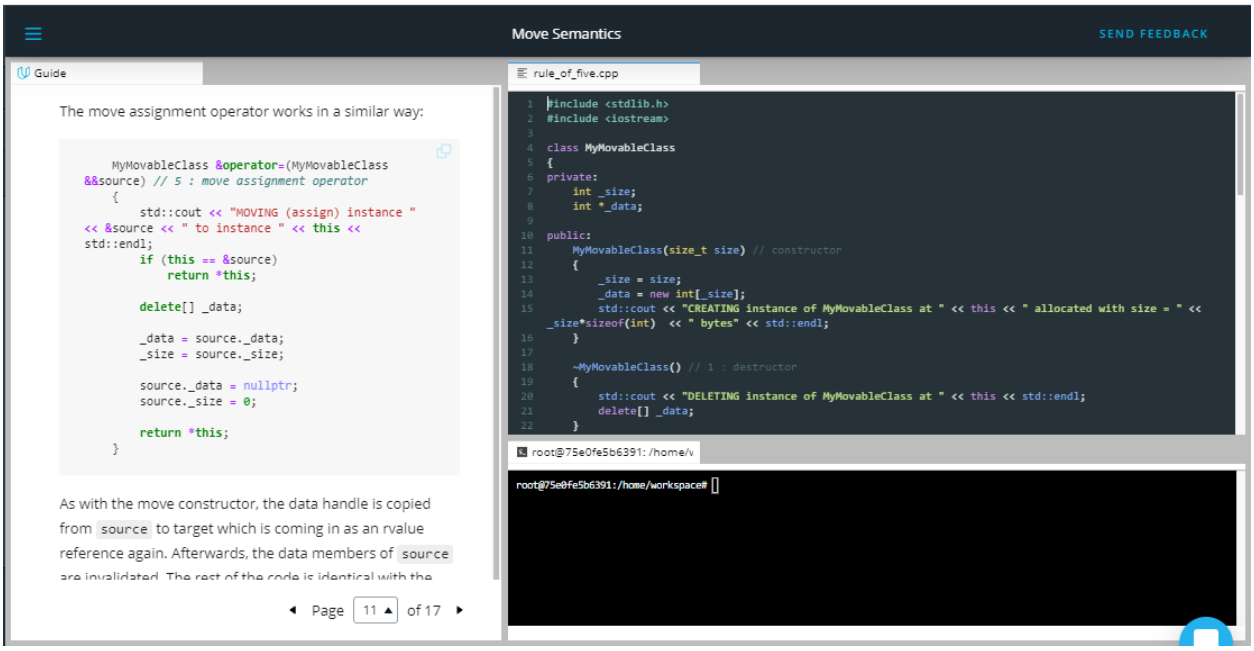
Just like the copy constructor, the move constructor builds an instance of a class using a source instance. The key difference between the two is that with the move constructor, the source instance will no longer be usable afterwards. Let us take a look at an implementation of the move constructor for our MyMovableClass:

```
MyMovableClass(MyMovableClass &&source) // 4 : move constructor
{
  std::cout << "MOVING (c'tor) instance " << &source << " to instance " << this << std::endl;
  _data = source._data;
  _size = source._size;
  source._data = nullptr;
  source._size = 0;
}
```

If you haven't already added it, you can add this code to the rule_of_five.cpp file to the right.

In this code, the move constructor takes as its input an rvalue reference to a source object of the same class. In doing so, we are able to use the object within the scope of the move constructor. As can be seen, the implementation copies the data handle from source to target and immediately invalidates source after copying is complete. Now, this is responsible for the data and must also release memory on

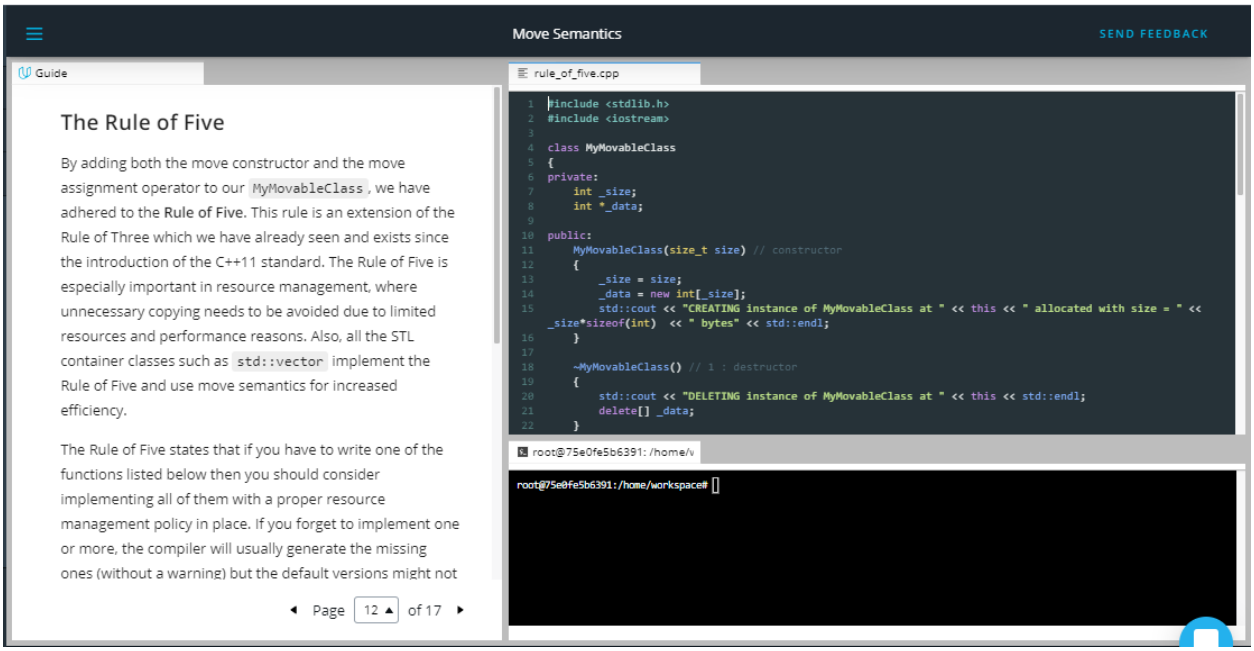
destruction - the ownership has been successfully changed (or moved) without the need to copy the data on the heap.



The move assignment operator works in a similar way:

```
MyMovableClass &operator=(MyMovableClass &&source) // 5 : move assignment operator
{
  std::cout << "MOVING (assign) instance " << &source << " to instance " << this << std::endl;
  if (this == &source)
    return *this;

  delete[] _data;

  _data = source._data;
  _size = source._size;

  source._data = nullptr;
  source._size = 0;

  return *this;
}
```

As with the move constructor, the data handle is copied from source to target which is coming in as an rvalue reference again. Afterwards, the data members of source are invalidated. The rest of the code is identical with the copy constructor we have already implemented.

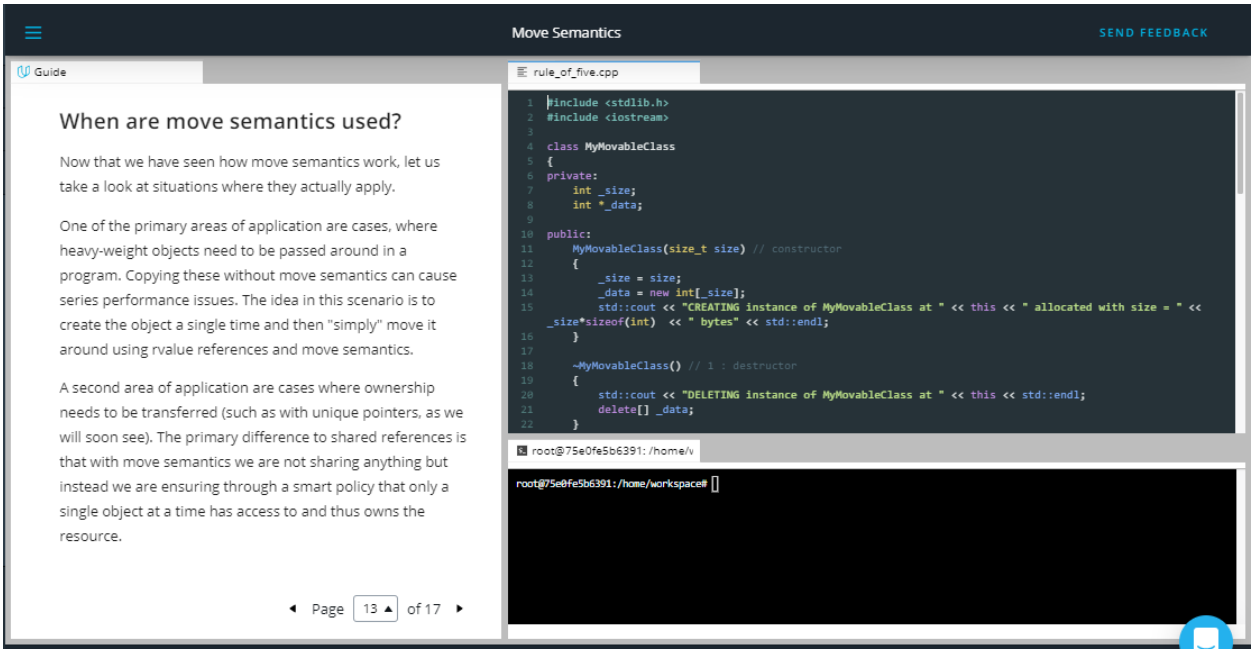


The Rule of Five

By adding both the move constructor and the move assignment operator to our MyMovableClass, we have adhered to the **Rule of Five**. This rule is an extension of the Rule of Three which we have already seen and exists since the introduction of the C++11 standard. The Rule of Five is especially important in resource management, where unnecessary copying needs to be avoided due to limited resources and performance reasons. Also, all the STL container classes such as std::vector implement the Rule of Five and use move semantics for increased efficiency.
The Rule of Five states that if you have to write one of the functions listed below then you should consider implementing all of them with a proper resource management policy in place. If you forget to implement one or more, the compiler will usually generate the missing ones (without a warning) but the default versions might not be suitable for the purpose you have in mind. The five functions are:

1. The **destructor**: Responsible for freeing the resource once the object it belongs to goes out of scope.
2. The **assignment operator**: The default assignment operation performs a member-wise shallow copy, which does not copy the content behind the resource handle. If a deep copy is needed, it has be implemented by the programmer.
3. The **copy constructor**: As with the assignment operator, the default copy constructor performs a shallow copy of the data members. If something else is needed, the programmer has to implement it accordingly.

4. The **move constructor**: Because copying objects can be an expensive operation which involves creating, copying and destroying temporary objects, rvalue references are used to bind to an rvalue. Using this mechanism, the move constructor transfers the ownership of a resource from a (temporary) rvalue object to a permanent lvalue object.
5. The **move assignment operator**: With this operator, ownership of a resource can be transferred from one object to another. The internal behavior is very similar to the move constructor.

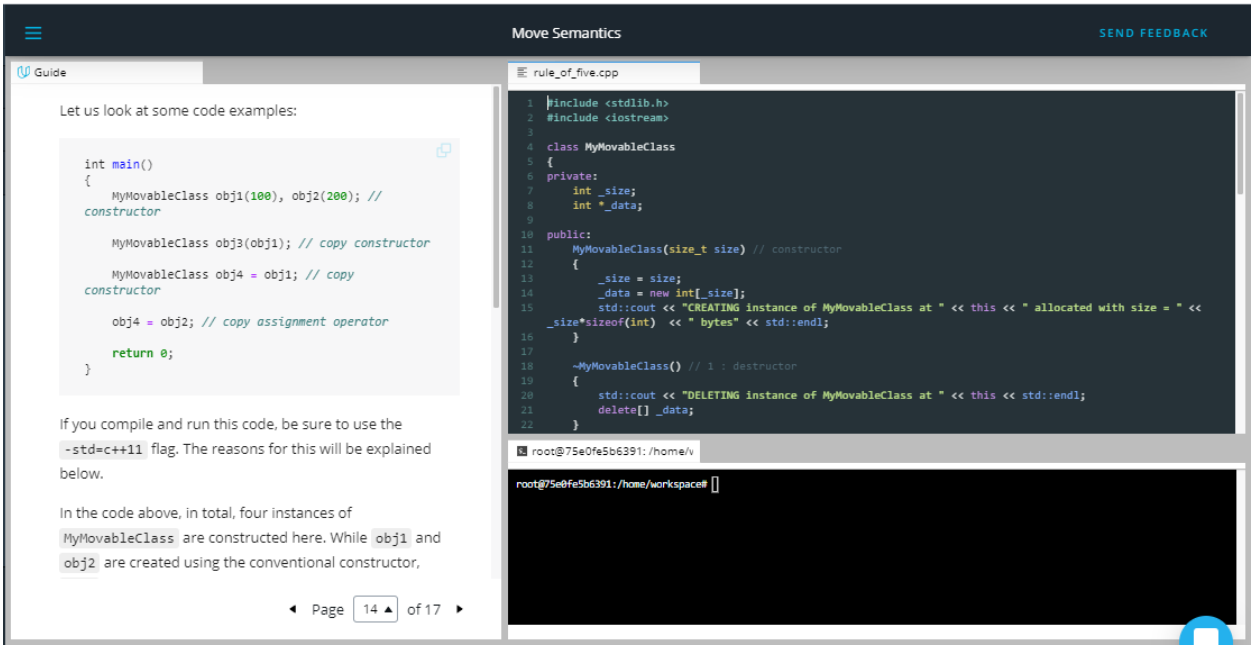


When are move semantics used?

Now that we have seen how move semantics work, let us take a look at situations where they actually apply.

One of the primary areas of application are cases, where heavy-weight objects need to be passed around in a program. Copying these without move semantics can cause series performance issues. The idea in this scenario is to create the object a single time and then "simply" move it around using rvalue references and move semantics.

A second area of application are cases where ownership needs to be transferred (such as with unique pointers, as we will soon see). The primary difference to shared references is that with move semantics we are not sharing anything but instead we are ensuring through a smart policy that only a single object at a time has access to and thus owns the resource.



Let us look at some code examples:

```
int main()
{
  MyMovableClass obj1(100), obj2(200); // constructor

  MyMovableClass obj3(obj1); // copy constructor

  MyMovableClass obj4 = obj1; // copy constructor

  obj4 = obj2; // copy assignment operator

  return 0;
}
```

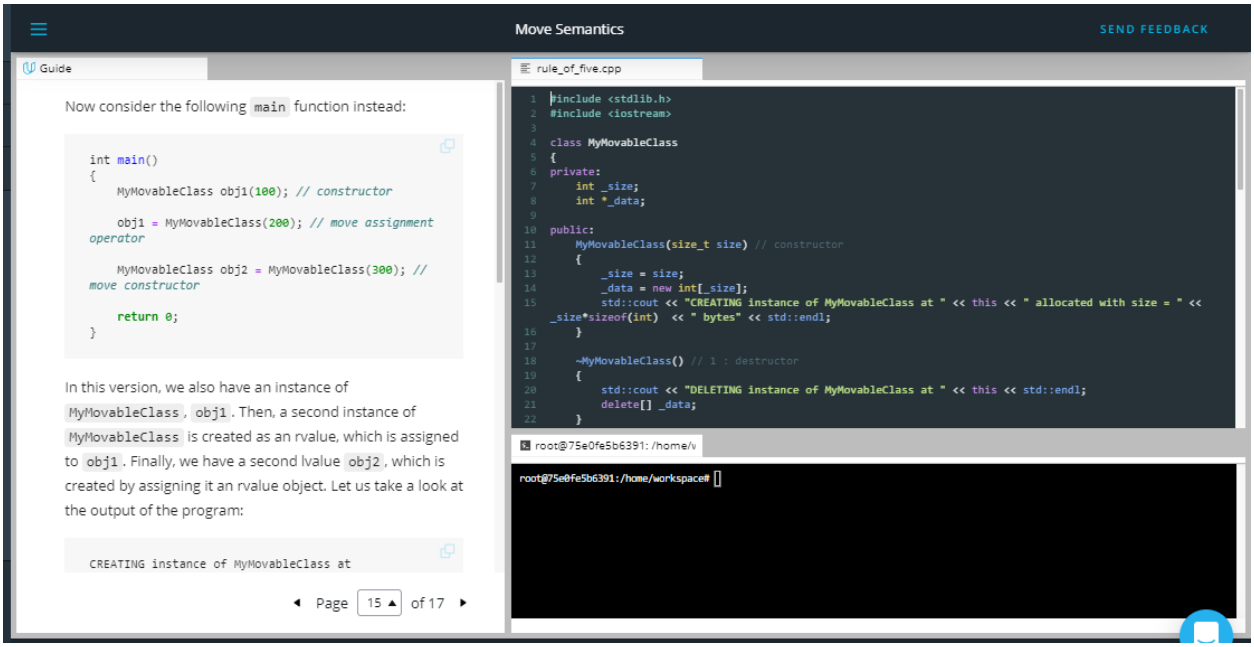
If you compile and run this code, be sure to use the -std=c++11 flag. The reasons for this will be explained below.

In the code above, in total, four instances of MyMovableClass are constructed here. While obj1 and obj2 are created using the conventional constructor, obj3 is created using the copy constructor instead according to our implementation. Interestingly, even though the creation of obj4 looks like an assignment, the compiler calls the copy constructor int this case. Finally, the last line calls the copy assignment operator. The output of the above main function looks like the following:

```
CREATING instance of MyMovableClass at 0x7ffeefbff718 allocated with size = 400 bytes

CREATING instance of MyMovableClass at 0x7ffeefbff708 allocated with size = 800 bytes

COPYING content of instance 0x7ffeefbff718 to instance 0x7ffeefbff6e8
```

```
COPYING content of instance 0x7ffeefbff718 to instance 0x7ffeefbff6d8

ASSIGNING content of instance 0x7ffeefbff708 to instance 0x7ffeefbff6d8

DELETING instance of MyMovableClass at 0x7ffeefbff6d8
DELETING instance of MyMovableClass at 0x7ffeefbff6e8
DELETING instance of MyMovableClass at 0x7ffeefbff708
DELETING instance of MyMovableClass at 0x7ffeefbff718
```

Note that the compiler has been called with the option `-fno-elide-constructors` to turn off an optimization techniques called *copy elision*, which would make it harder to understand the various calls and the operations they entail. This technique is guaranteed to be used as of C++17, which is why we are also reverting to the C++11 standard for the remainder of this chapter using `-std=c++11`. Until now, no move operation has been performed yet as all of the above calls were involving lvalues.



Now consider the following `main` function instead:

```
int main()
{
  MyMovableClass obj1(100); // constructor

  obj1 = MyMovableClass(200); // move assignment operator

  MyMovableClass obj2 = MyMovableClass(300); // move constructor

  return 0;
}
```

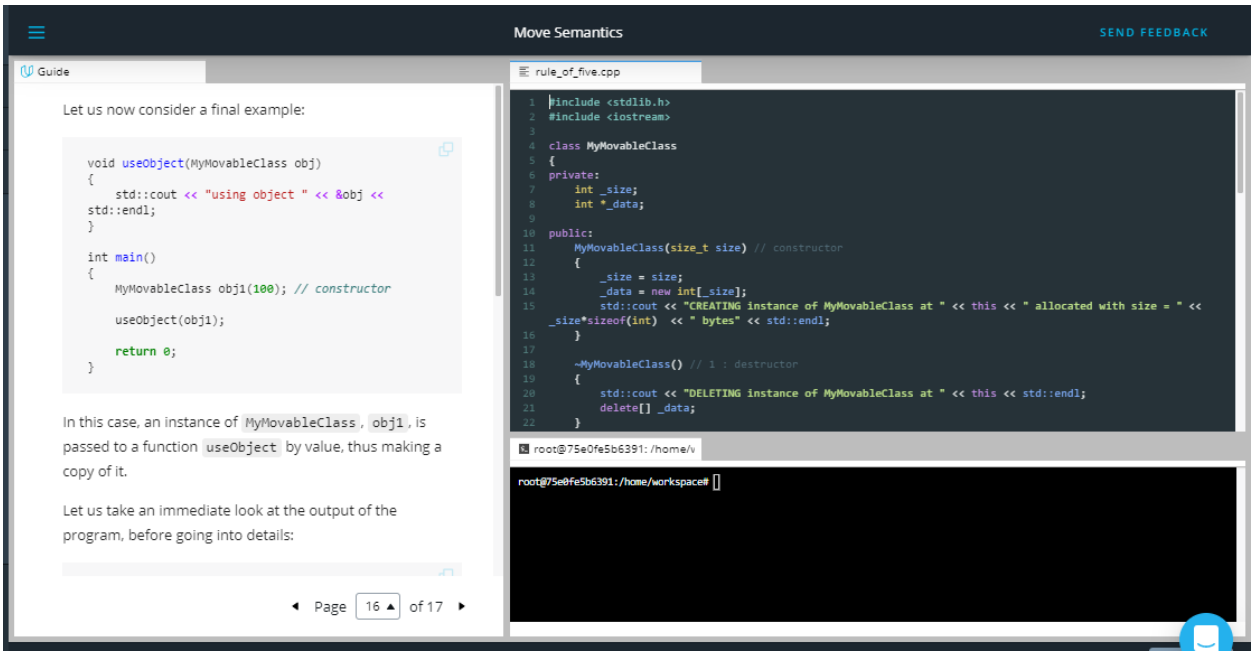
In this version, we also have an instance of `MyMovableClass`, `obj1`. Then, a second instance of `MyMovableClass` is created as an rvalue, which is assigned to `obj1`. Finally, we

have a second lvalue obj2, which is created by assigning it an rvalue object. Let us take a look at the output of the program:

```
CREATING instance of MyMovableClass at 0x7ffeefbff718 allocated with size = 400 bytes

CREATING instance of MyMovableClass at 0x7ffeefbff708 allocated with size = 800 bytes

MOVING (assign) instance 0x7ffeefbff708 to instance 0x7ffeefbff718

DELETING instance of MyMovableClass at 0x7ffeefbff708

CREATING instance of MyMovableClass at 0x7ffeefbff6d8 allocated with size = 1200 bytes

MOVING (c'tor) instance 0x7ffeefbff6d8 to instance 0x7ffeefbff6e8

DELETING instance of MyMovableClass at 0x7ffeefbff6d8
DELETING instance of MyMovableClass at 0x7ffeefbff6e8
DELETING instance of MyMovableClass at 0x7ffeefbff718
```

By looking at the stack addresses of the objects, we can see that the temporary object at 0x7ffeefbff708 is moved to 0x7ffeefbff718 using the move assignment operator we wrote earlier, because the instance obj1 is assigned an rvalue. As expected from an rvalue, its destructor is called immediately afterwards. But as we have made sure to null its data pointer in the move constructor, the actual data will not be deleted. The advantage from a performance perspective in this case is that no deep-copy of the rvalue object needs to be made, we are simply redirecting the internal resource handle thus making an efficient shallow copy.

Next, another temporary instance with a size of 1200 bytes is created as a temporary object and "assigned" to obj3. Note that while the call looks like an assignment, the move constructor is called under the hood, making the call identical to `MyMovableClass obj2(MyMovableClass(300));`. By creating obj3 in such a way, we are reusing the temporary rvalue and transferring ownership of its resources to the newly created obj3.



Let us now consider a final example:

```
void useObject(MyMovableClass obj)
{
  std::cout << "using object " << &obj << std::endl;
}

int main()
{
  MyMovableClass obj1(100); // constructor

  useObject(obj1);

  return 0;
}
```

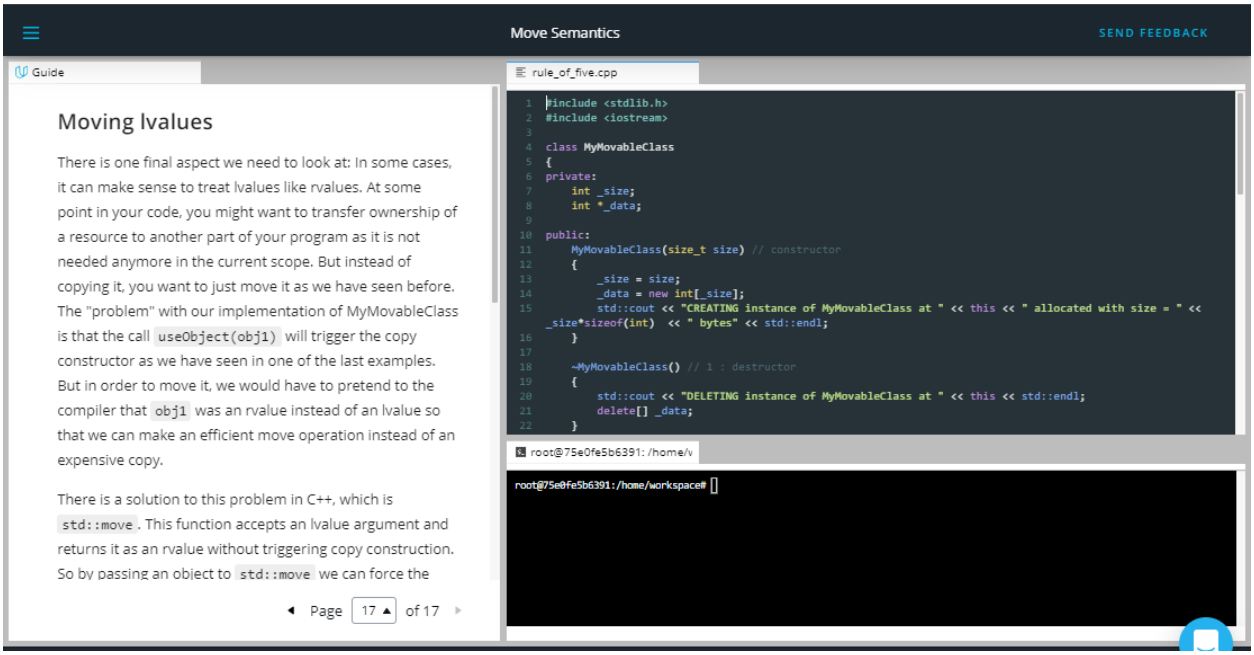
In this case, an instance of `MyMovableClass`, `obj1`, is passed to a function `useObject` by value, thus making a copy of it.

Let us take an immediate look at the output of the program, before going into details:

```
(1)
CREATING instance of MyMovableClass at 0x7ffeefbff718 allocated with size = 400 bytes

(2)
COPYING content of instance 0x7ffeefbff718 to instance 0x7ffeefbff708

using object 0x7ffeefbff708

(3)
DELETING instance of MyMovableClass at 0x7ffeefbff708
```

```
(4)
CREATING instance of MyMovableClass at 0x7ffeefbff6d8 allocated with size = 800 bytes

(5)
MOVING (c'tor) instance 0x7ffeefbff6d8 to instance 0x7ffeefbff6e8

using object 0x7ffeefbff6e8

DELETING instance of MyMovableClass at 0x7ffeefbff6e8
DELETING instance of MyMovableClass at 0x7ffeefbff6d8
DELETING instance of MyMovableClass at 0x7ffeefbff718
```

First, we are creating an instance of MyMovableClass, `obj1`, by calling the constructor of the class (1).
Then, we are passing `obj1` by-value to a function `useObject`, which causes a temporary object `obj` to be instantiated, which is a copy of `obj1` (2) and is deleted immediately after the function scope is left (3).
Then, the function is called with a temporary instance of `MyMovableClass` as its argument, which creates a temporary instance of `MyMovableClass` as an rvalue (4). But instead of making a copy of it as before, the move constructor is used (5) to transfer ownership of that temporary object to the function scope, which saves us one expensive deep-copy.



Moving lvalues

There is one final aspect we need to look at: In some cases, it can make sense to treat lvalues like rvalues. At some point in your code, you might want to transfer ownership of a resource to another part of your program as it is not needed anymore in the current scope. But instead of copying it, you want to just move it as we have seen before. The "problem" with our implementation of MyMovableClass is that the call `useObject(obj1)` will trigger the copy constructor as we have seen in one of the last examples. But in order to move it, we would have to pretend to the compiler that `obj1` was an rvalue instead of an lvalue so that we can make an efficient move operation instead of an expensive copy.
There is a solution to this problem in C++, which is `std::move`. This function accepts an lvalue argument and returns it as an rvalue without triggering copy construction. So by passing an object to `std::move` we can force the compiler to use move semantics, either in the form of move constructor or the move assignment operator:

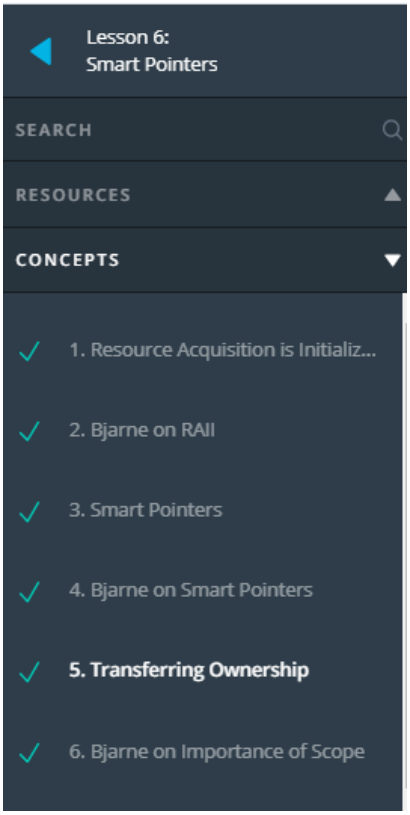
```
int main()
{
  MyMovableClass obj1(100); // constructor

  useObject(std::move(obj1));

  return 0;
}
```

Nothing much has changed, apart from `obj1` being passed to the `std::move` function. The output would look like the following:

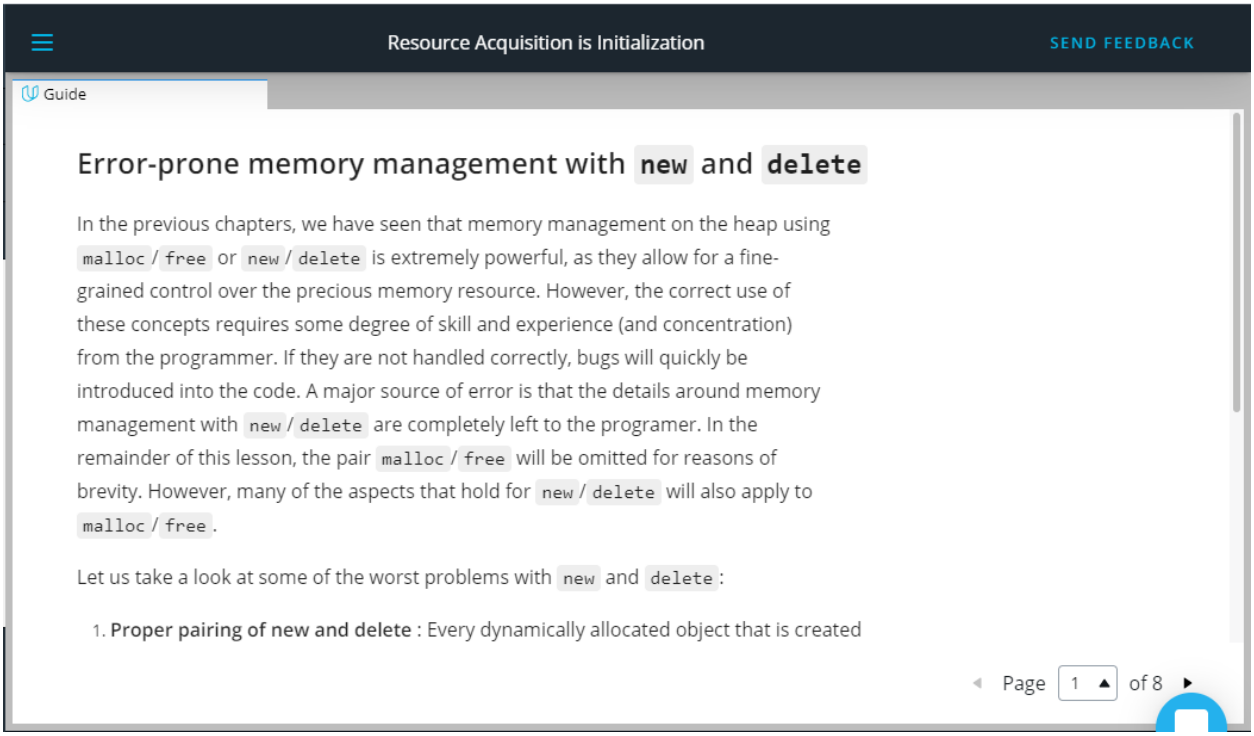
```
CREATING instance of MyMovableClass at 0x7ffeefbff718 allocated with size = 400 bytes

MOVING (c'tor) instance 0x7ffeefbff718 to instance 0x7ffeefbff708

using object 0x7ffeefbff708

DELETING instance of MyMovableClass at 0x7ffeefbff708
DELETING instance of MyMovableClass at 0x7ffeefbff718
```

By using std::move, we were able to pass the ownership of the resources within `obj1` to the function `useObject`. The local copy `obj1` in the argument list was created with the move constructor and thus accepted the ownership transfer from `obj1` to `obj` . Note that after the call to `useObject`, the instance `obj1` has been invalidated by setting its internal handle to null and thus may not be used anymore within the scope of `main` (even though you could theoretically try to access it, but this would be a really bad idea).

Outro
https://youtu.be/QR279x7G9pA



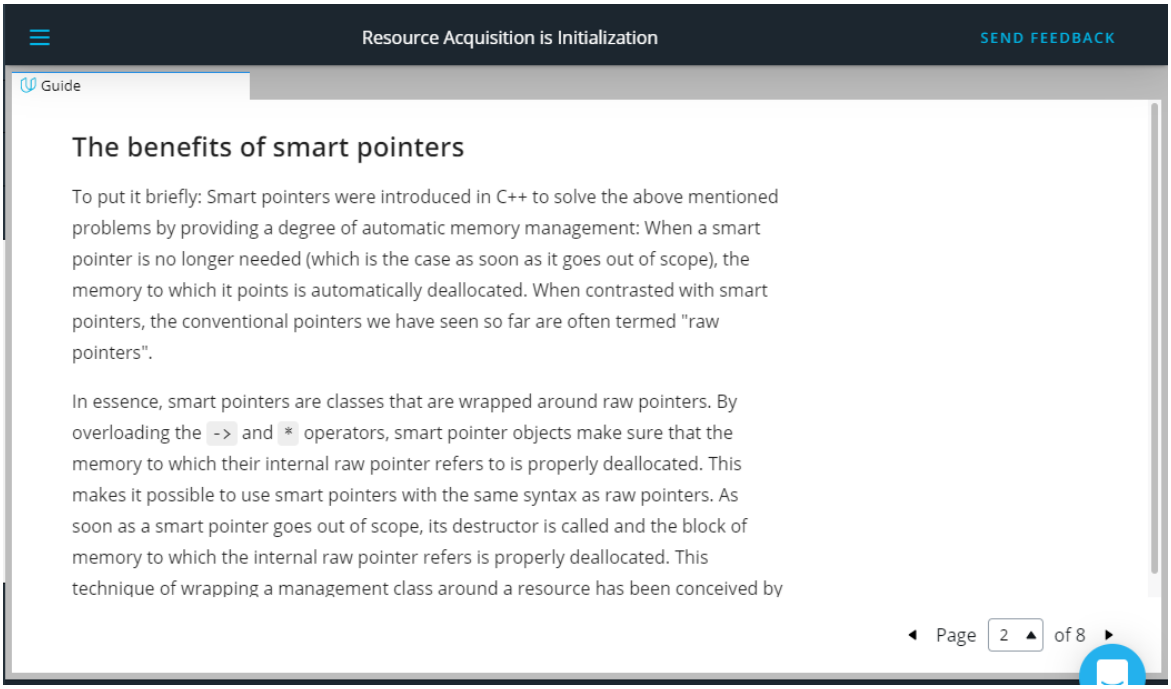
1) Resource Acquisition is Initialization
https://youtu.be/ScH7IDXJ8d0



Error-prone memory management with new and delete

In the previous chapters, we have seen that memory management on the heap using malloc/free or new/delete is extremely powerful, as they allow for a fine-grained control over the precious memory resource. However, the correct use of these concepts requires some degree of skill and experience (and concentration) from the programmer. If they are not handled correctly, bugs will quickly be introduced into the code. A major source of error is that the details around memory management with new/delete are completely left to the programer. In the remainder of this lesson, the pair malloc/free will be omitted for reasons of brevity. However, many of the aspects that hold for new/delete will also apply to malloc/free.
Let us take a look at some of the worst problems with new and delete:

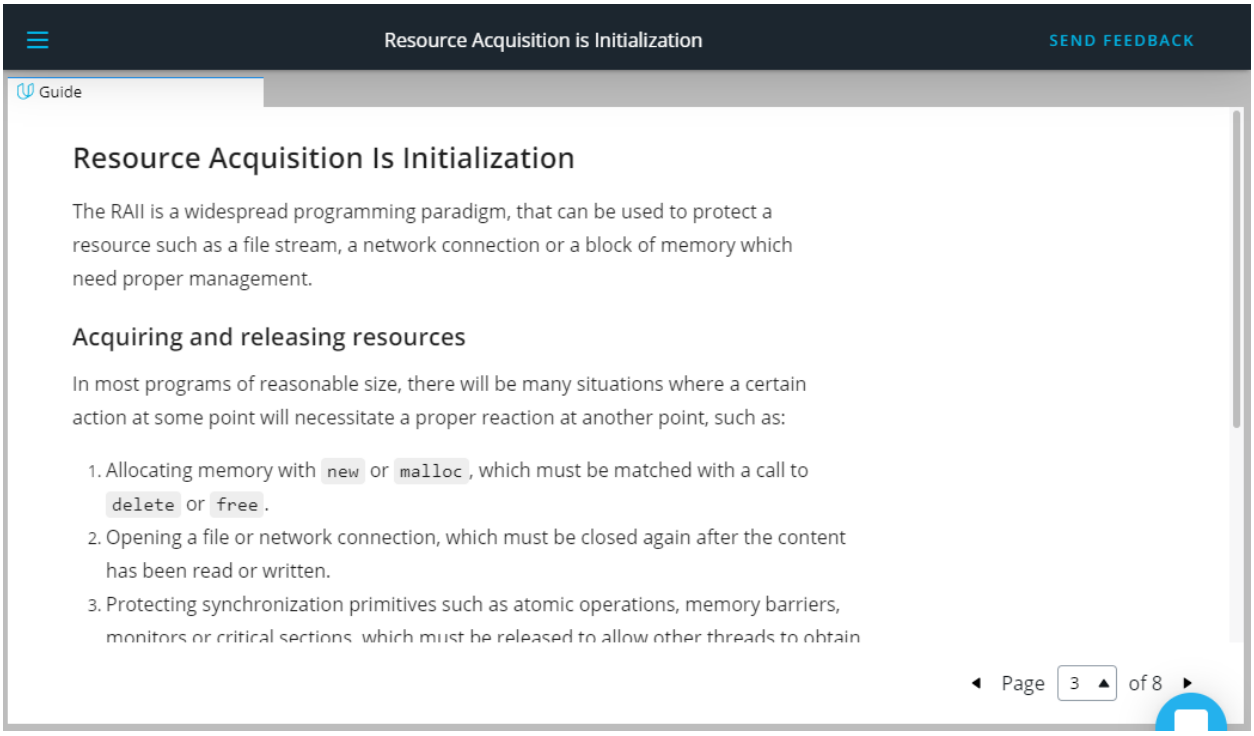
1. **Proper pairing of new and delete** : Every dynamically allocated object that is created with new must be followed by a manual deallocation at a "proper" place in the program. If the programer forgets to call delete (which can happen very quickly) or if it is done at an "inappropriate" position, memory leaks will occur which might clog up a large portion of memory.
2. **Correct operator pairing** : C++ offers a variety of new/delete operators, especially when dealing with arrays on the heap. A dynamically allocated array initialized with new[] may only be deleted with the operator delete[]. If the wrong operator is used, program behavior will be undefined - which is to be avoided at all cost in C++.
3. **Memory ownership** : If a third-party function returns a pointer to a data structure, the only way of knowing who will be responsible for resource deallocation is by looking into either the code or the documentation. If both are not available (as is often the case), there is no way to infer the ownership from the return type. As an example, in the final project of this course, we will use the graphical library wxWidgets to create the user interface of a chatbot application. In wxWidgets, the programmer can create child windows and control elements on the heap using new, but the framework will take care of deletion altogether. If for some reason the programmer



The benefits of smart pointers

To put it briefly: Smart pointers were introduced in C++ to solve the above mentioned problems by providing a degree of automatic memory management: When a smart pointer is no longer needed (which is the case as soon as it goes out of scope), the memory to which it points is automatically deallocated. When contrasted with smart pointers, the conventional pointers we have seen so far are often termed "raw pointers".

In essence, smart pointers are classes that are wrapped around raw pointers. By overloading the -> and * operators, smart pointer objects make sure that the memory to which their internal raw pointer refers to is properly deallocated. This makes it possible to use smart pointers with the same syntax as raw pointers. As soon as a smart pointer goes out of scope, its destructor is called and the block of memory to which the internal raw pointer refers is properly deallocated. This technique of wrapping a management class around a resource has been conceived by Bjarne Stroustroup and is called **Resource Acquisition Is Initialization (RAII)**. Before



Resource Acquisition Is Initialization

The RAII is a widespread programming paradigm, that can be used to protect a resource such as a file stream, a network connection or a block of memory which need proper management.

Acquiring and releasing resources

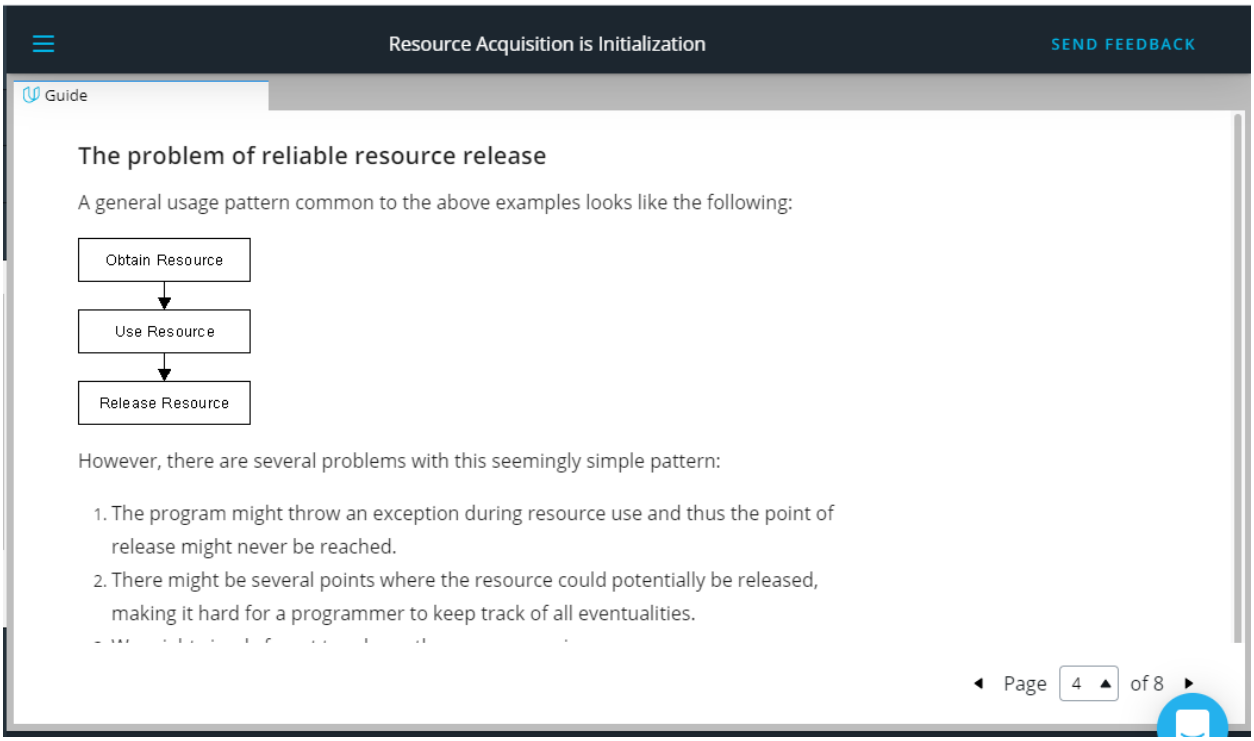
In most programs of reasonable size, there will be many situations where a certain action at some point will necessitate a proper reaction at another point, such as:

1. Allocating memory with new or malloc, which must be matched with a call to delete or free.

2. Opening a file or network connection, which must be closed again after the content has been read or written.
3. Protecting synchronization primitives such as atomic operations, memory barriers, monitors or critical sections, which must be released to allow other threads to obtain them.

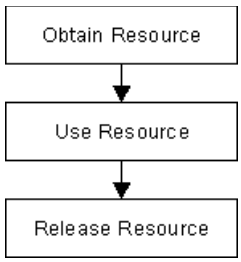
The following table gives a brief overview of some resources and their respective allocation and deallocation calls in C++:

	Acquire	Release
Files	`fopen`	`fclose`
Memory	`new, new[]`	`delete, delete[]`
Locks	`lock, try_lock`	`unlock`



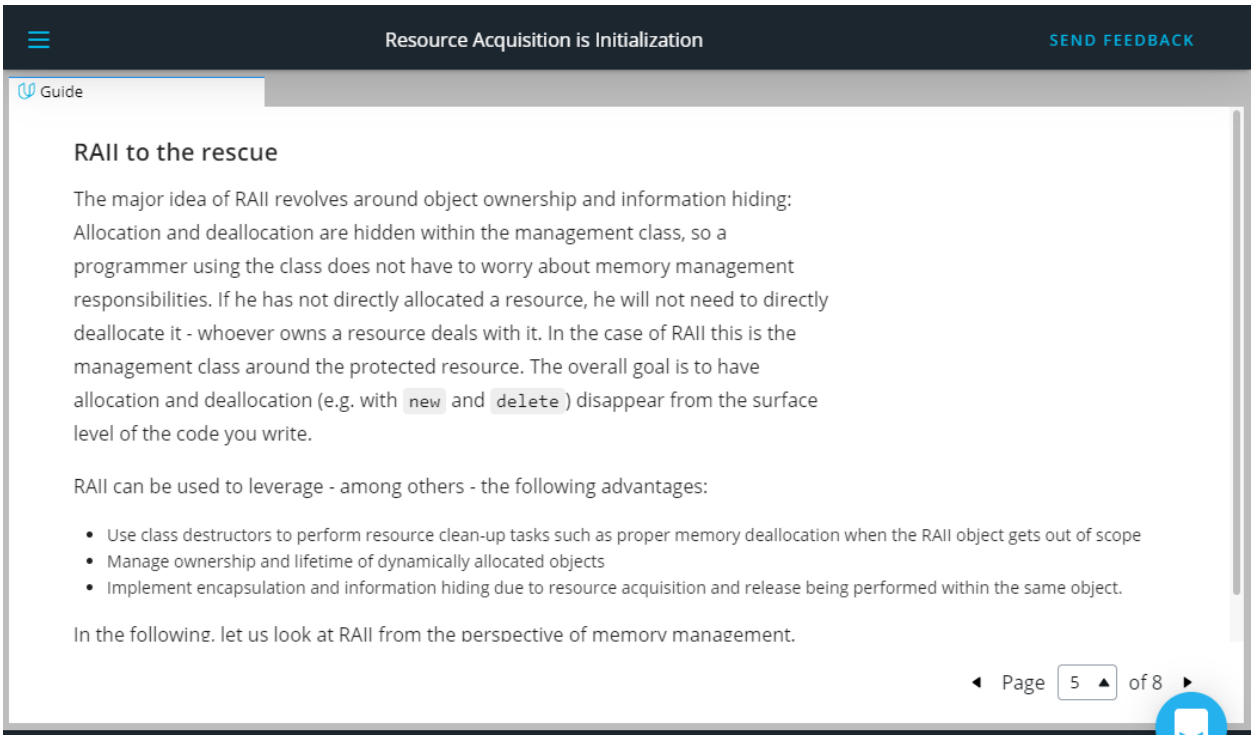
The problem of reliable resource release¶

A general usage pattern common to the above examples looks like the following:



However, there are several problems with this seemingly simple pattern:

1. The program might throw an exception during resource use and thus the point of release might never be reached.
2. There might be several points where the resource could potentially be released, making it hard for a programmer to keep track of all eventualities.
3. We might simply forget to release the resource again.



RAII to the rescue

The major idea of RAII revolves around object ownership and information hiding: Allocation and deallocation are hidden within the management class, so a programmer using the class does not have to worry about memory management responsibilities. If he has not directly allocated a resource, he will not need to directly deallocate it - whoever owns a resource deals with it. In the case of RAII this is the management class around the protected resource. The overall goal is to have allocation and deallocation (e.g. with `new` and `delete`) disappear from the surface level of the code you write.

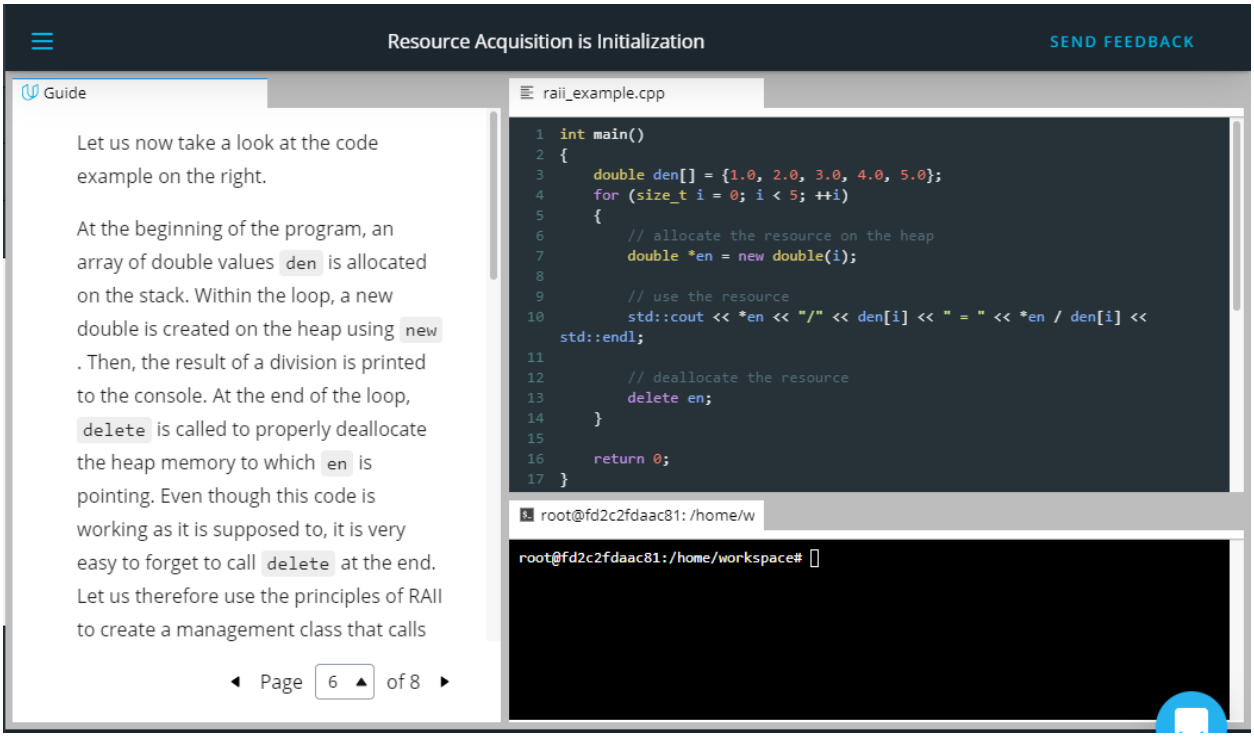
RAII can be used to leverage - among others - the following advantages:

- Use class destructors to perform resource clean-up tasks such as proper memory deallocation when the RAII object gets out of scope
- Manage ownership and lifetime of dynamically allocated objects

- Implement encapsulation and information hiding due to resource acquisition and release being performed within the same object.

In the following, let us look at RAII from the perspective of memory management. There are three major parts to an RAII class:

1. A resource is allocated in the constructor of the RAII class
2. The resource is deallocated in the destructor
3. All instances of the RAII class are allocated on the stack to reliably control the lifetime via the object scope



Let us now take a look at the code example on the right.

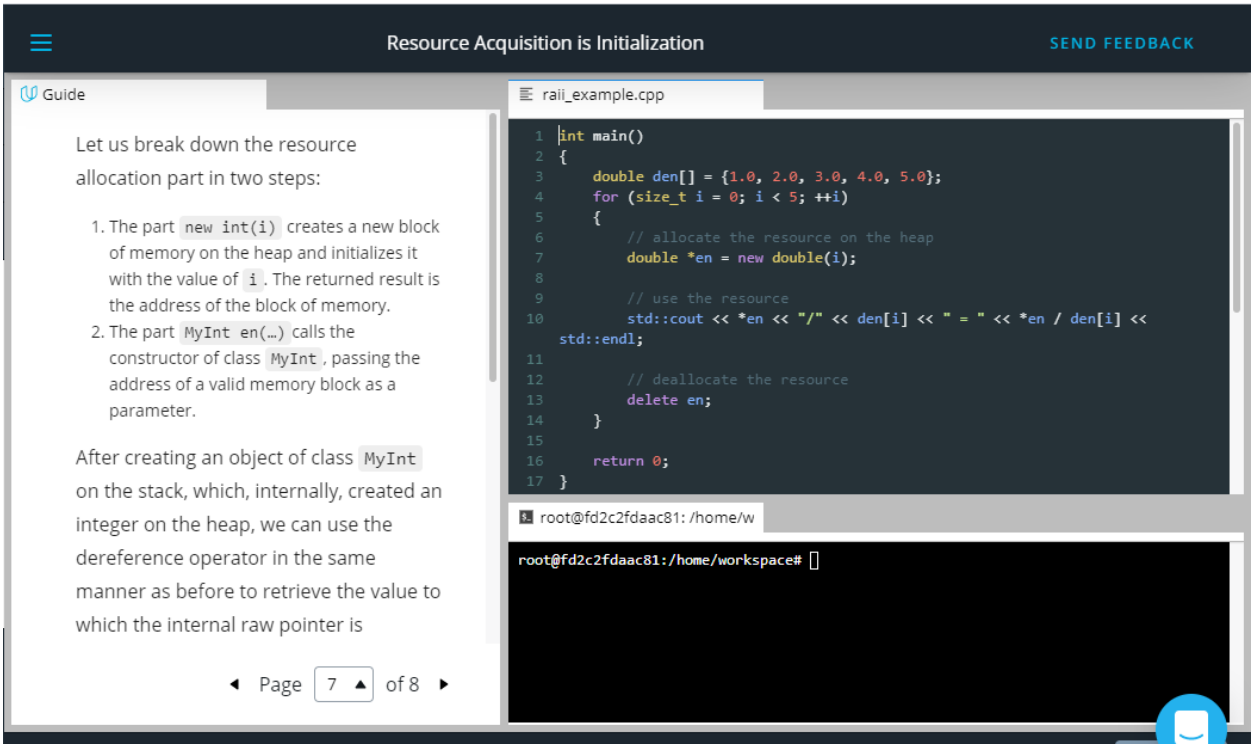
At the beginning of the program, an array of double values den is allocated on the stack. Within the loop, a new double is created on the heap using new. Then, the result of a division is printed to the console. At the end of the loop, delete is called to properly deallocate the heap memory to which en is pointing. Even though this code is working as it is supposed to, it is very easy to forget to call delete at the end. Let us therefore use the principles of RAII to create a management class that calls delete automatically:

```
class MyInt
{
  int *_p; // pointer to heap data
public:
  MyInt(int *p = NULL) { _p = p; }
```

```
  ~MyInt()
  {
    std::cout << "resource " << *_p << " deallocated" << std::endl;
    delete _p;
  }
  int &operator*() { return *_p; } /// / / overload dereferencing operator
};
```

In this example, the constructor of class `MyInt` takes a pointer to a memory resource. When the destructor of a `MyInt` object is called, the resource is deleted from memory - which makes `MyInt` an RAII memory management class. Also, the `*` operator is overloaded which enables us to dereference `MyInt` objects in the same manner as with raw pointers. Let us therefore slightly alter our code example from above to see how we can properly use this new construct:

```
int main()
{
  double den[] = {1.0, 2.0, 3.0, 4.0, 5.0};
  for (size_t I = 0; I < 5; ++i)
  {
    // allocate the resource on the stack
    MyInt en(new int(i));

    // use the resource
    std::cout << *en << "/" << den[i] << " = " << *en / den[i] << std::endl;
  }

  return 0;
}
```



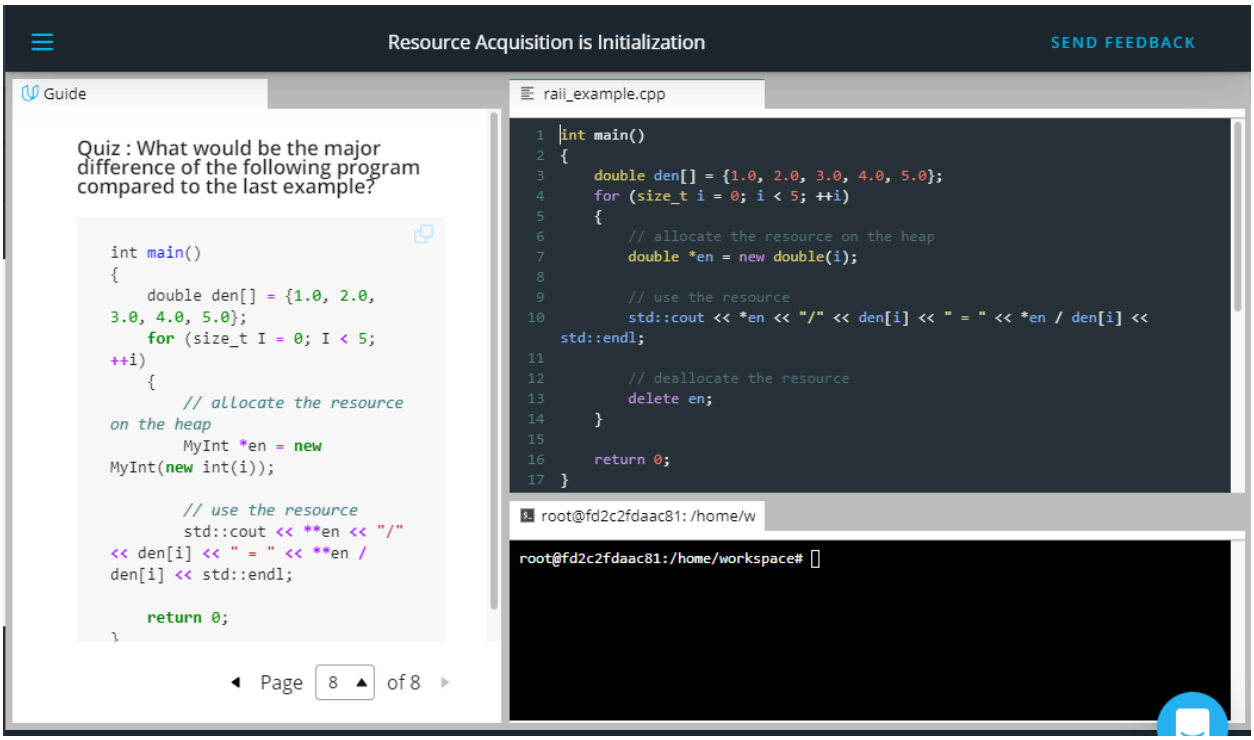
Let us break down the resource allocation part in two steps:

1. The part `new int(i)` creates a new block of memory on the heap and initializes it with the value of `i`. The returned result is the address of the block of memory.
2. The part `MyInt en(...)` calls the constructor of class `MyInt`, passing the address of a valid memory block as a parameter.

After creating an object of class `MyInt` on the stack, which, internally, created an integer on the heap, we can use the dereference operator in the same manner as before to retrieve the value to which the internal raw pointer is pointing. Because the `MyInt` object `en` lives on the stack, it is automatically deallocated after each loop cycle - which automatically calls the destructor to release the heap memory. The following console output verifies this:

```
0/1 = 0
resource 0 deallocated
1/2 = 0.5
resource 1 deallocated
2/3 = 0.666667
resource 2 deallocated
3/4 = 0.75
resource 3 deallocated
4/5 = 0.8
resource 4 deallocated
```

We have thus successfully used the RAII idiom to create a memory management class that spares us from thinking about calling delete. By creating the `MyInt` object on the



Quiz : What would be the major difference of the following program compared to the last example?

```
int main()
{
  double den[] = {1.0, 2.0, 3.0, 4.0, 5.0};
  for (size_t I = 0; I < 5; ++i)
  {
    // allocate the resource on the heap
    MyInt *en = new MyInt(new int(i));

    // use the resource
    std::cout << **en << "/" << den[i] << " = " << **en / den[i] << std::endl;

  return 0;
}
```

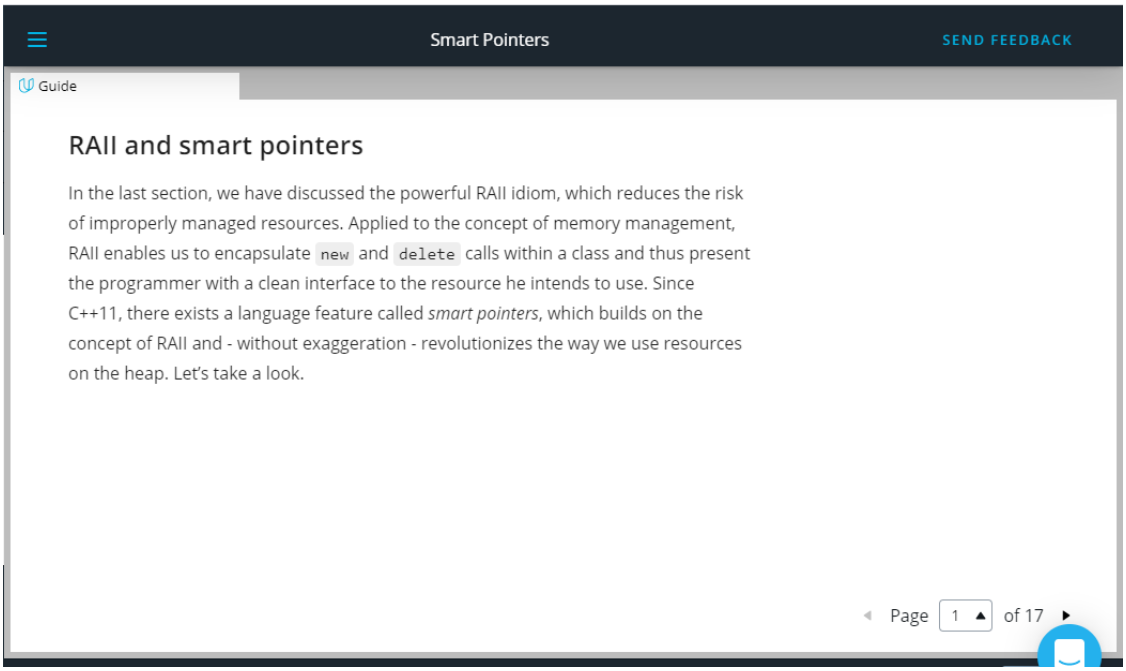
HIDE SOLUTION

ANSWER: The destructor of `MyInt` would never be called, hence causing a memory leak with each loop iteration.

Outro
https://youtu.be/qCZT8tKatrU

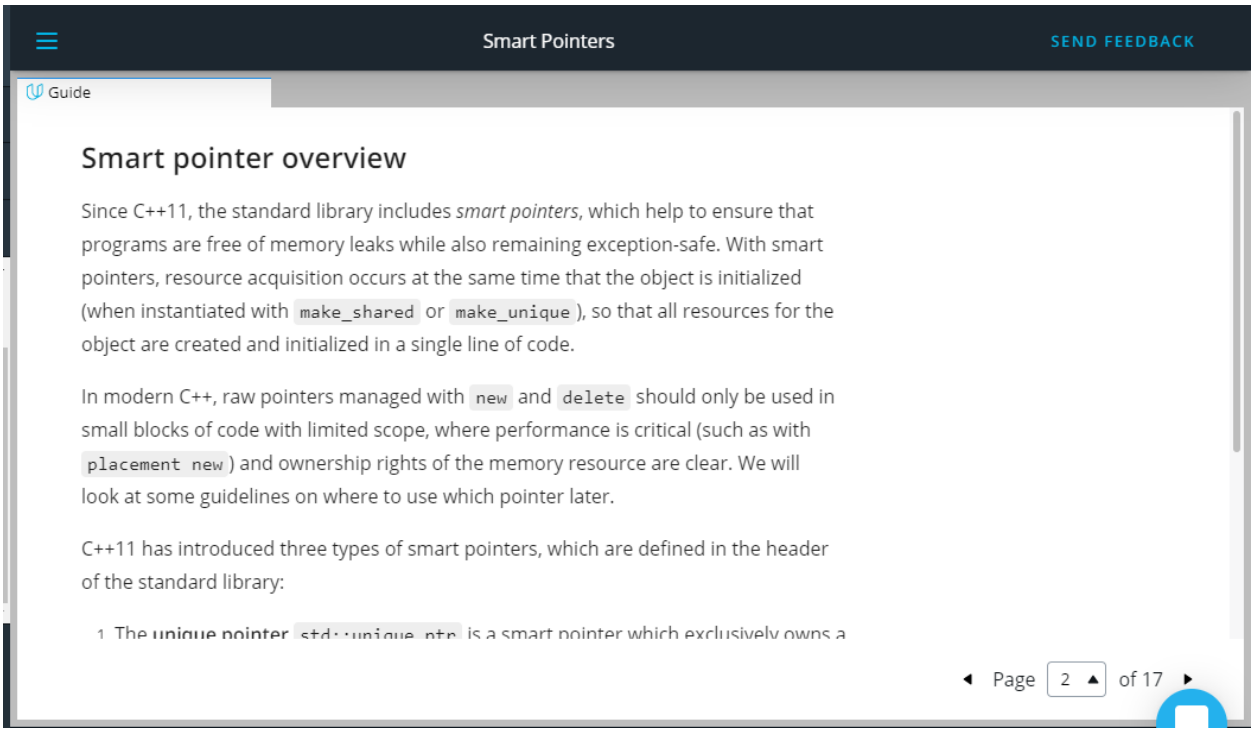
2) Bjarne on RAII
https://youtu.be/eSCgbrpSNj0

3) Smart Pointers
https://youtu.be/Nr3qzOklQNk



RAII and smart pointers

In the last section, we have discussed the powerful RAII idiom, which reduces the risk of improperly managed resources. Applied to the concept of memory management, RAII enables us to encapsulate `new` and `delete` calls within a class and thus present the programmer with a clean interface to the resource he intends to use. Since C++11, there exists a language feature called *smart pointers*, which builds on the concept of RAII and - without exaggeration - revolutionizes the way we use resources on the heap. Let's take a look.



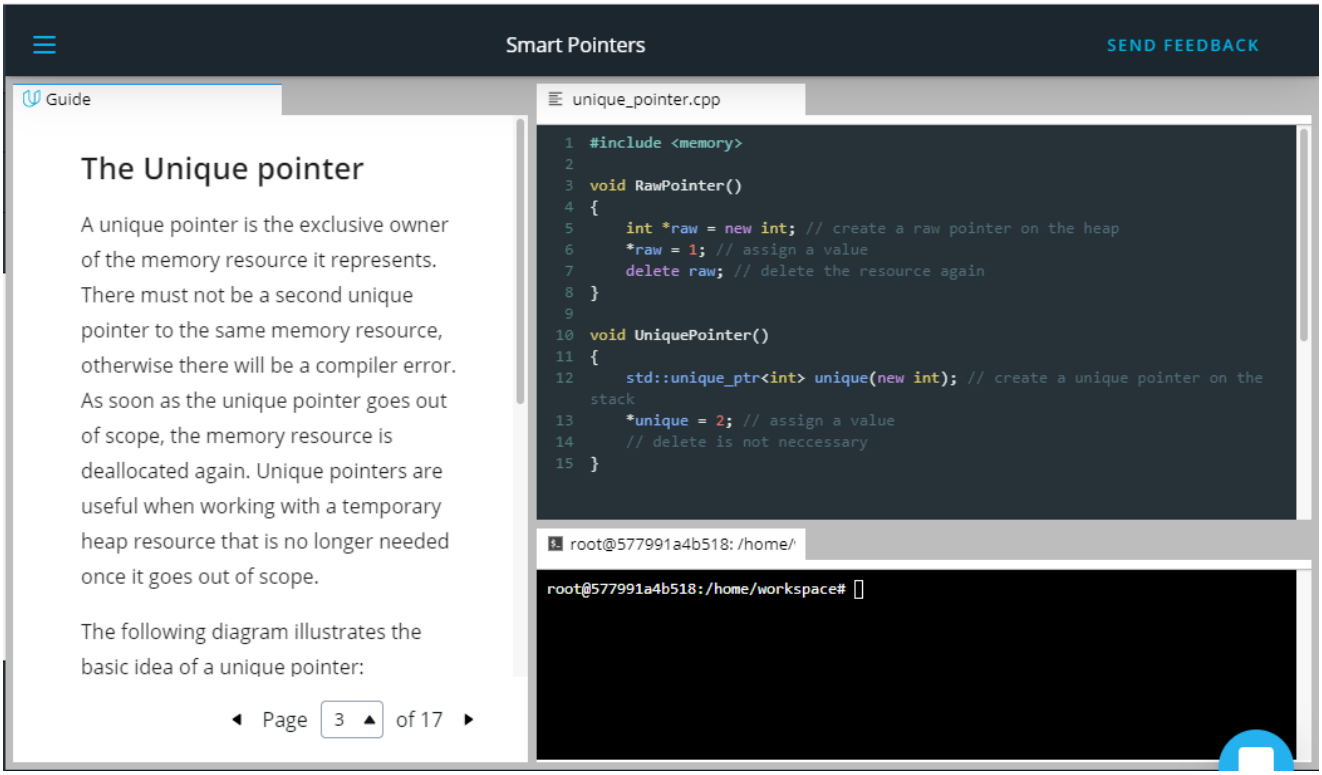
Smart pointer overview

Since C++11, the standard library includes *smart pointers*, which help to ensure that programs are free of memory leaks while also remaining exception-safe. With smart pointers, resource acquisition occurs at the same time that the object is initialized (when instantiated with `make_shared` or `make_unique`), so that all resources for the object are created and initialized in a single line of code.
In modern C++, raw pointers managed with `new` and `delete` should only be used in small blocks of code with limited scope, where performance is critical (such as with `placement new`) and ownership rights of the memory resource are clear. We will look at some guidelines on where to use which pointer later.
C++11 has introduced three types of smart pointers, which are defined in the header of the standard library:

1. The **unique pointer** `std::unique_ptr` is a smart pointer which exclusively owns a dynamically allocated resource on the heap. There must not be a second unique pointer to the same resource.

2. The **shared pointer** std::shared_ptr points to a heap resource but does not explicitly own it. There may even be several shared pointers to the same resource, each of which will increase an internal reference count. As soon as this count reaches zero, the resource will automatically be deallocated.
3. The **weak pointer** std::weak_ptr behaves similar to the shared pointer but does not increase the reference counter.

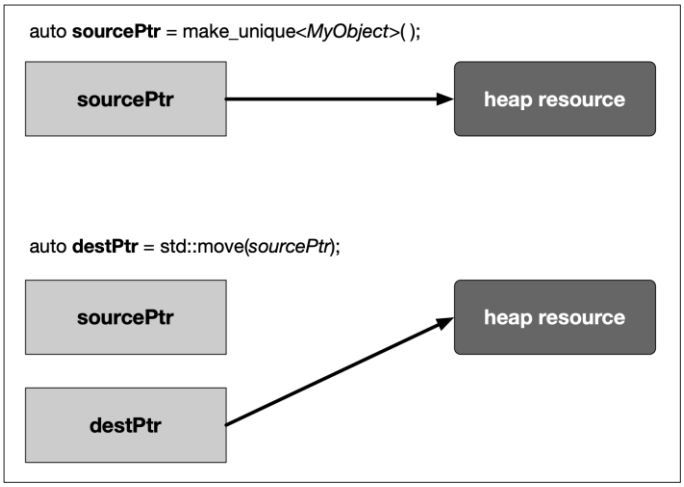
Prior to C++11, there was a concept called std::auto_ptr, which tried to realize a similar idea. However, this concept can now be safely considered as deprecated and should not be used anymore.
Let us now look at each of the three smart pointer types in detail.



The Unique pointer

A unique pointer is the exclusive owner of the memory resource it represents. There must not be a second unique pointer to the same memory resource, otherwise there will be a compiler error. As soon as the unique pointer goes out of scope, the memory resource is deallocated again. Unique pointers are useful when working with a temporary heap resource that is no longer needed once it goes out of scope.

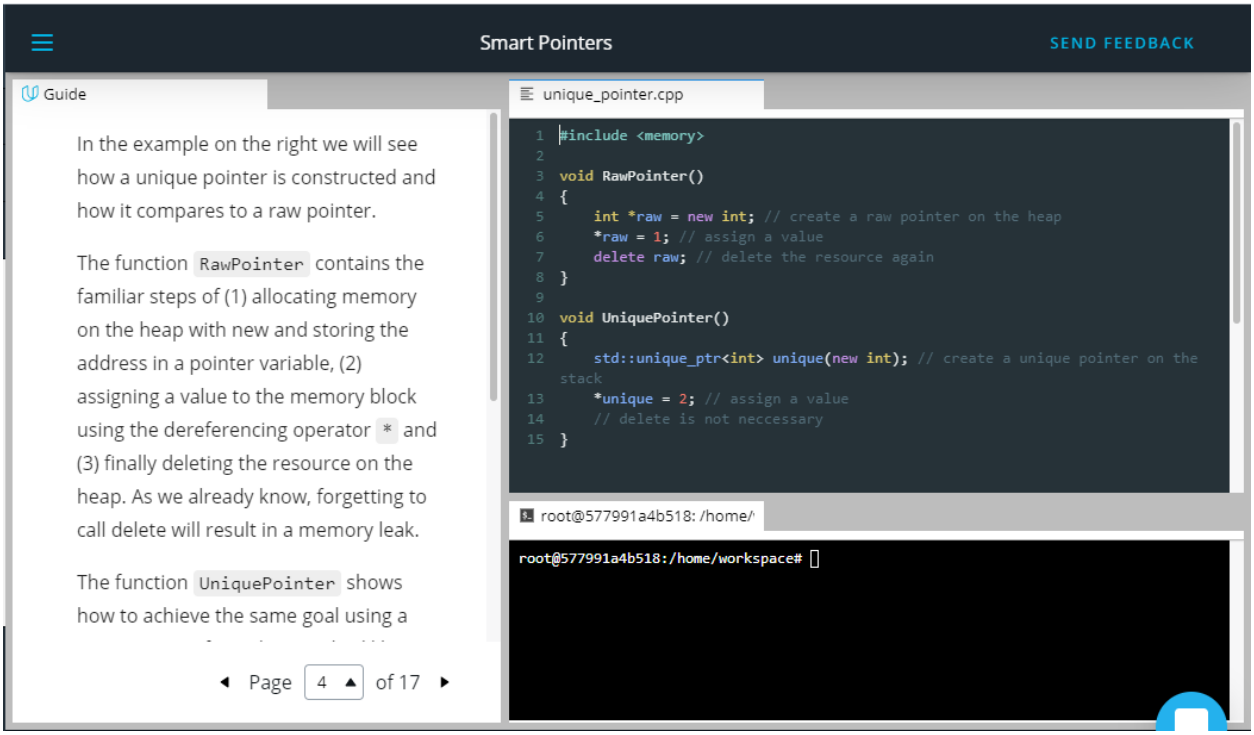
The following diagram illustrates the basic idea of a unique pointer:



In the example, a resource in memory is referenced by a unique pointer instance sourcePtr. Then, the resource is reassigned to another unique pointer instance destPtr using std::move. The resource is now owned by destPtr while sourcePtr can still be used but does not manage a resource anymore.

A unique pointer is constructed using the following syntax:

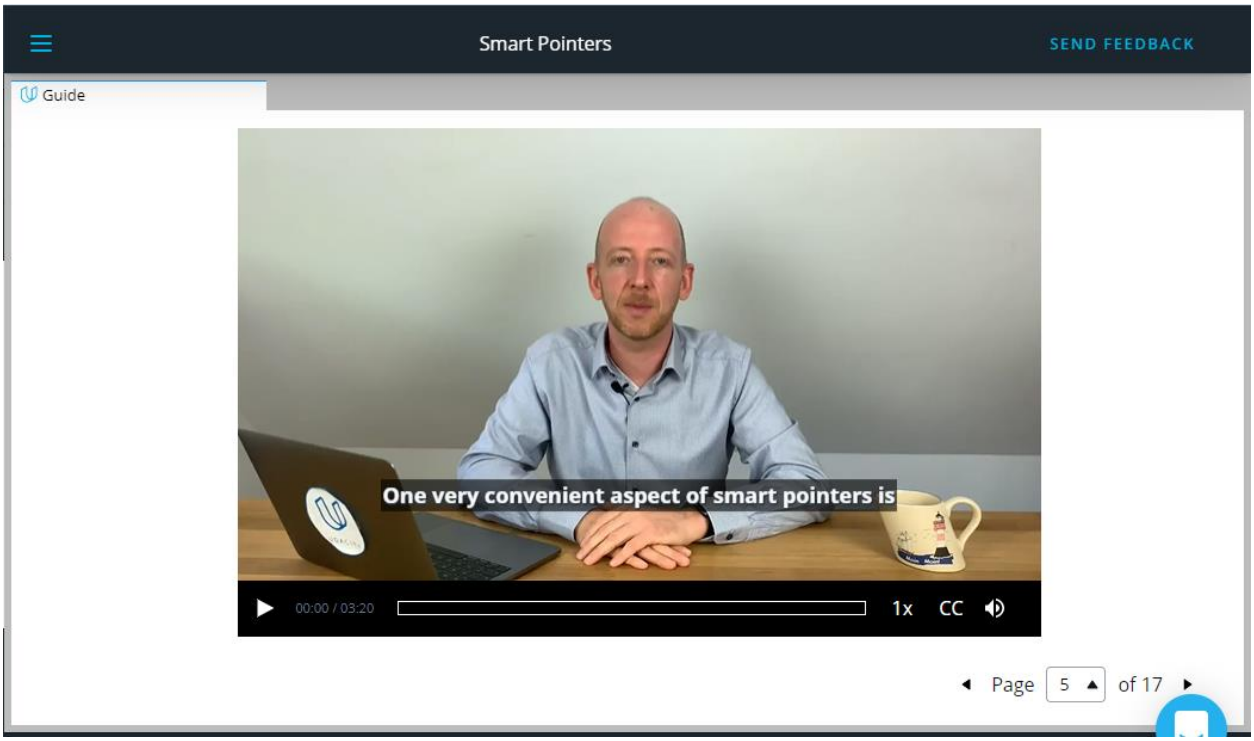
std::unique_ptr<Type> p(new Type);



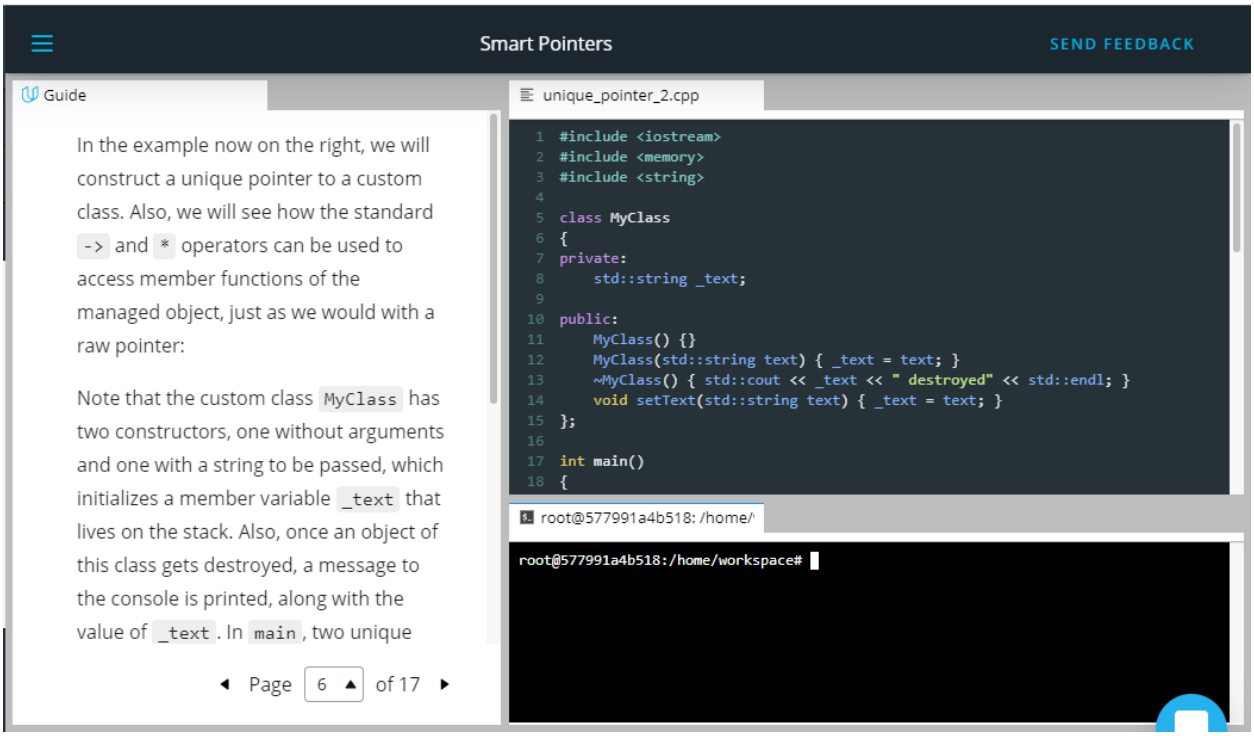
In the example on the right we will see how a unique pointer is constructed and how it compares to a raw pointer.

The function RawPointer contains the familiar steps of (1) allocating memory on the heap with new and storing the address in a pointer variable, (2) assigning a value to the memory block using the

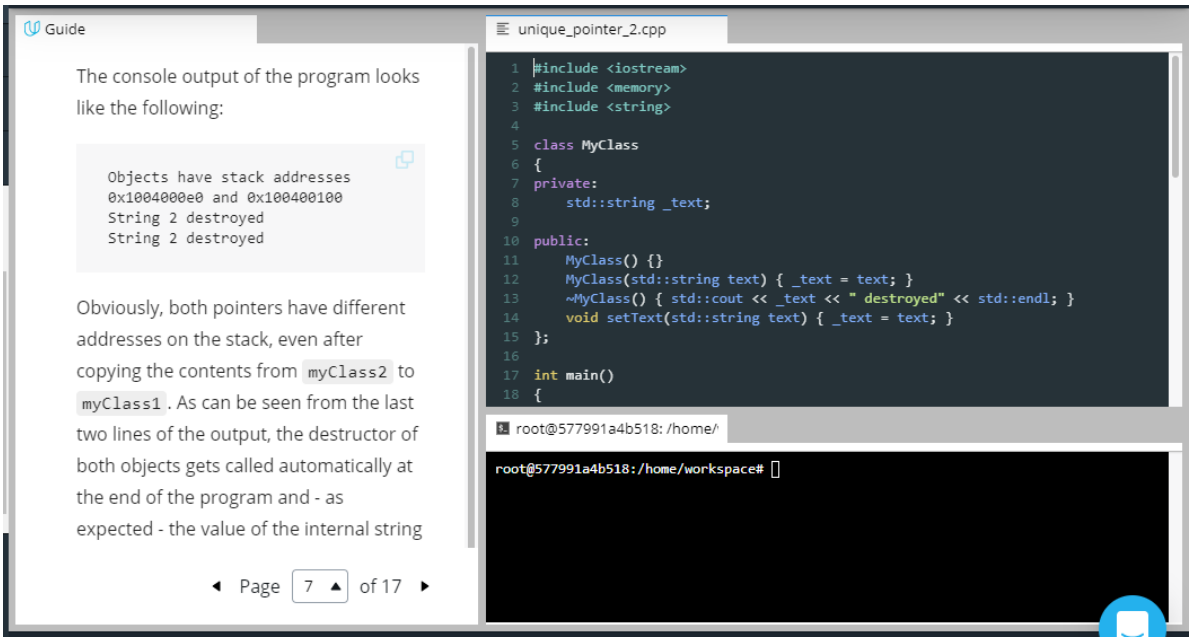
dereferencing operator * and (3) finally deleting the resource on the heap. As we already know, forgetting to call delete will result in a memory leak.
The function `UniquePointer` shows how to achieve the same goal using a smart pointer from the standard library. As can be seen, a smart pointer is a class template that is declared on the stack and then initialized by a raw pointer (returned by `new`) to a heap-allocated object. The smart pointer is now responsible for deleting the memory that the raw pointer specifies - which happens as soon as the smart pointer goes out of scope. Note that smart pointers always need to be declared on the stack, otherwise the scoping mechanism would not work.
The smart pointer destructor contains the call to delete, and because the smart pointer is declared on the stack, its destructor is invoked when the smart pointer goes out of scope, even if an exception is thrown.



https://video.udacity-data.com/topher/2019/September/5d865440_nd213-c03-l05-02.2-smart-pointers-sc/nd213-c03-l05-02.2-smart-pointers-sc_720p.mp4



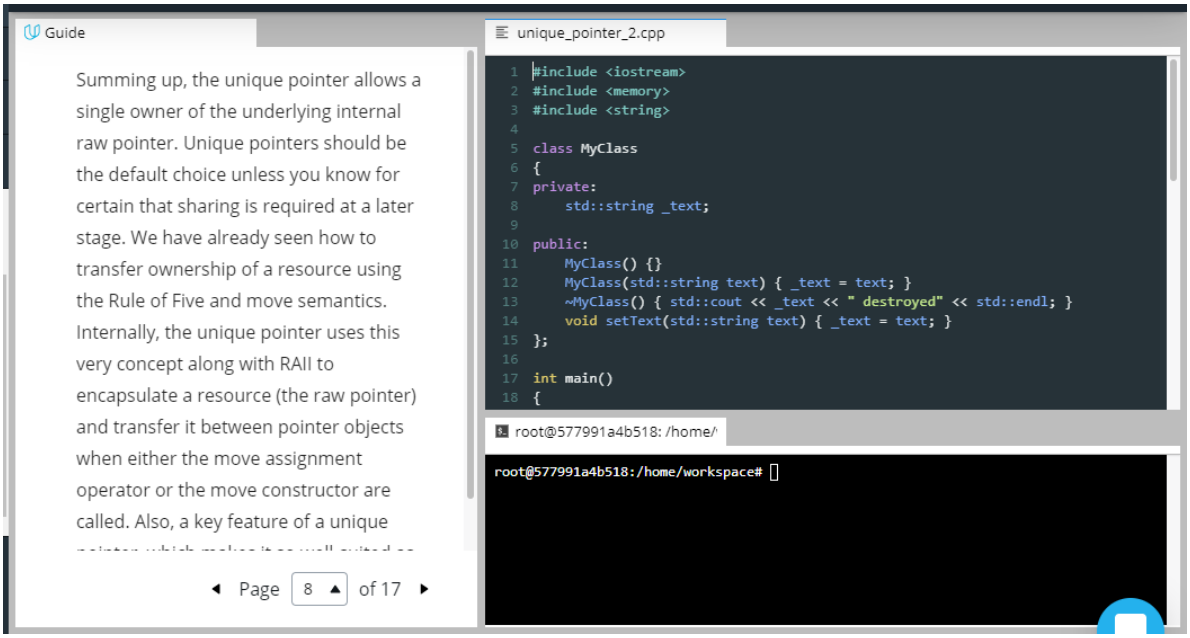
In the example now on the right, we will construct a unique pointer to a custom class. Also, we will see how the standard -> and * operators can be used to access member functions of the managed object, just as we would with a raw pointer:
Note that the custom class MyClass has two constructors, one without arguments and one with a string to be passed, which initializes a member variable _text that lives on the stack. Also, once an object of this class gets destroyed, a message to the console is printed, along with the value of _text. In main, two unique pointers are created with the address of a MyClass object on the heap as arguments. With myClass2, we can see that constructor arguments can be passed just as we would with raw pointers. After both pointers have been created, we can use the -> operator to access members of the class, such as calling the function setText. From looking at the function call alone you would not be able to tell that myClass1 is in fact a smart pointer. Also, we can use the dereference operator * to access the value of myClass1 and myClass2 and assign the one to the other. Finally, the . operator gives us access to proprietary functions of the smart pointer, such as retrieving the internal raw pointer with get().



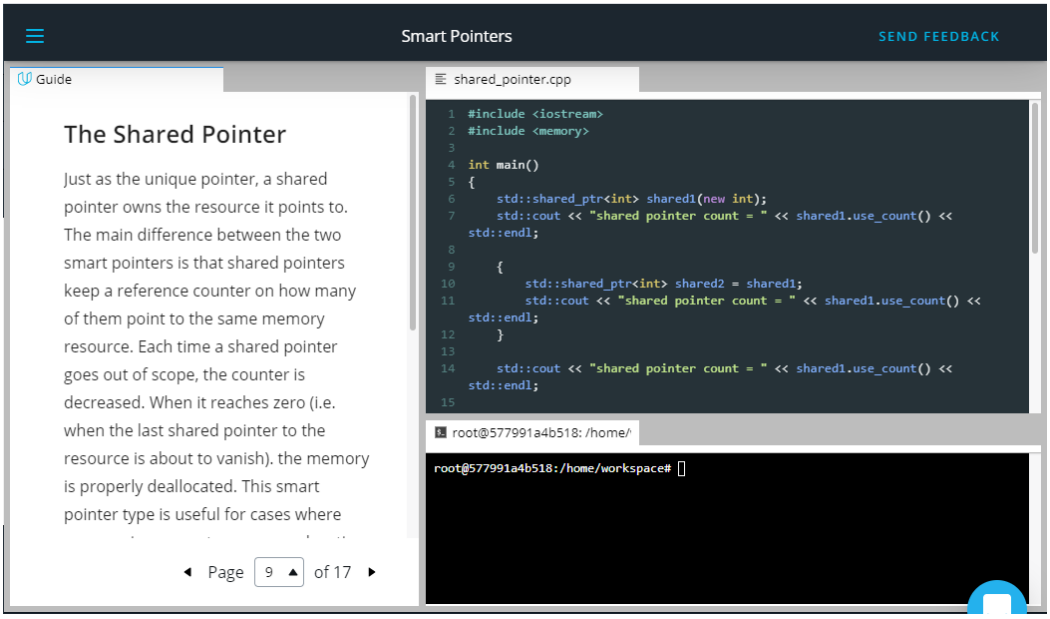
The console output of the program looks like the following:

```
Objects have stack addresses 0x1004000e0 and 0x100400100
String 2 destroyed
String 2 destroyed
```

Obviously, both pointers have different addresses on the stack, even after copying the contents from myClass2 to myClass1. As can be seen from the last two lines of the output, the destructor of both objects gets called automatically at the end of the program and - as expected - the value of the internal string is identical due to the copy operation.



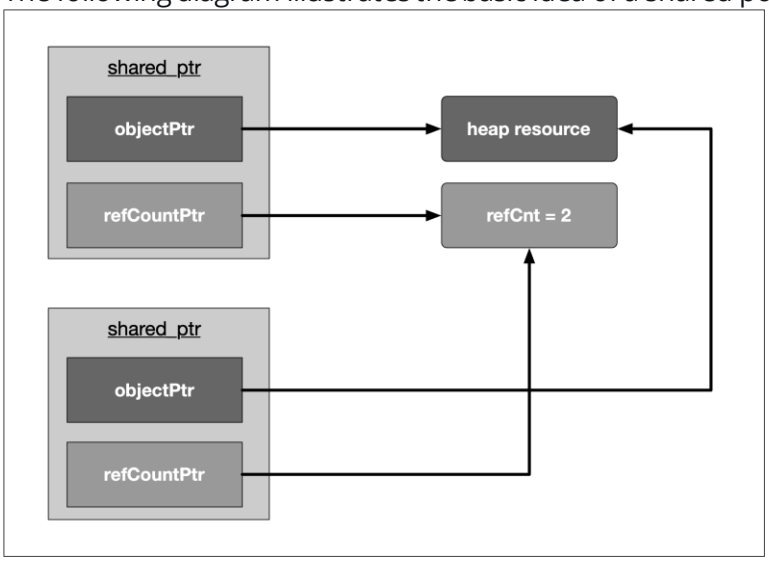
Summing up, the unique pointer allows a single owner of the underlying internal raw pointer. Unique pointers should be the default choice unless you know for certain that sharing is required at a later stage. We have already seen how to transfer ownership of a resource using the Rule of Five and move semantics. Internally, the unique pointer uses this very concept along with RAII to encapsulate a resource (the raw pointer) and transfer it between pointer objects when either the move assignment operator or the move constructor are called. Also, a key feature of a unique pointer, which makes it so well-suited as a return type for many functions, is the possibility to convert it to a shared pointer. We will have a deeper look into this in the section on ownership transfer.



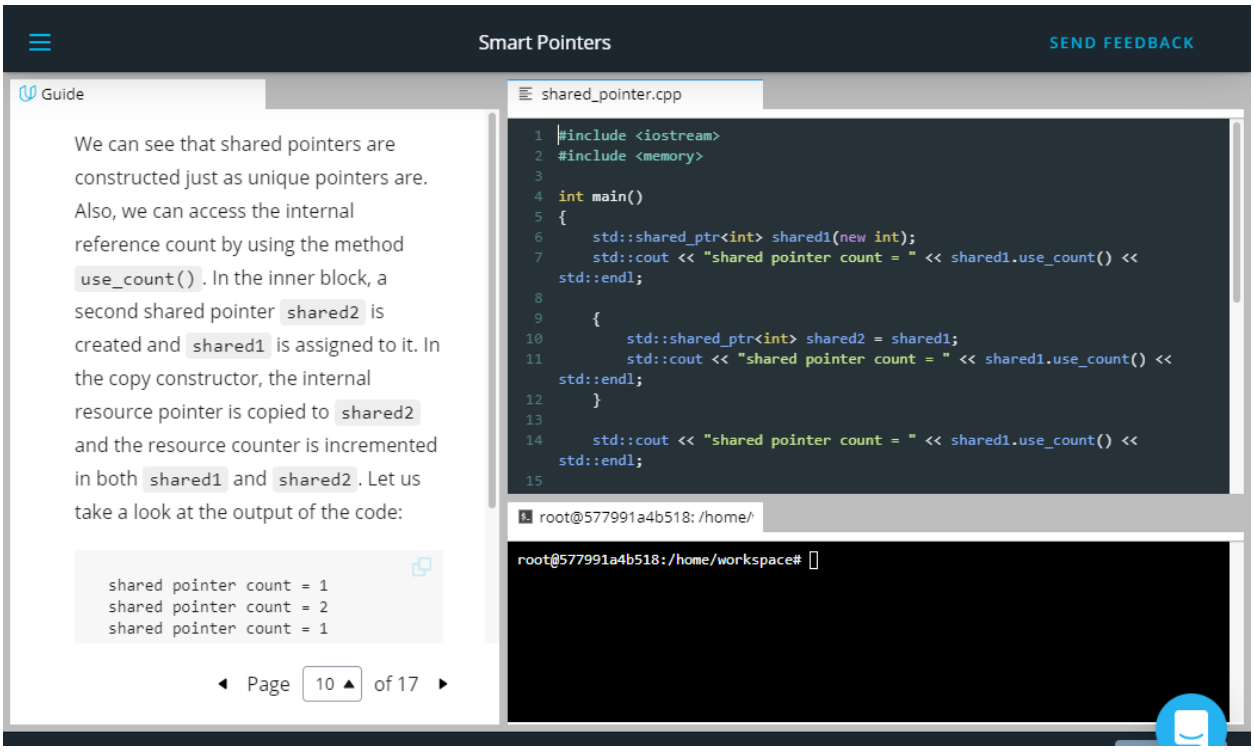
The Shared Pointer

Just as the unique pointer, a shared pointer owns the resource it points to. The main difference between the two smart pointers is that shared pointers keep a reference counter on how many of them point to the same memory resource. Each time a shared pointer goes out of scope, the counter is decreased. When it reaches zero (i.e. when the last shared pointer to the resource is about to vanish). the memory is properly deallocated. This smart pointer type is useful for cases where you require access to a memory location on the heap in multiple parts of your program and you want to make sure that whoever owns a shared pointer to the memory can rely on the fact that it will be accessible throughout the lifetime of that pointer.

The following diagram illustrates the basic idea of a shared pointer:



Please take a look at the code on the right.

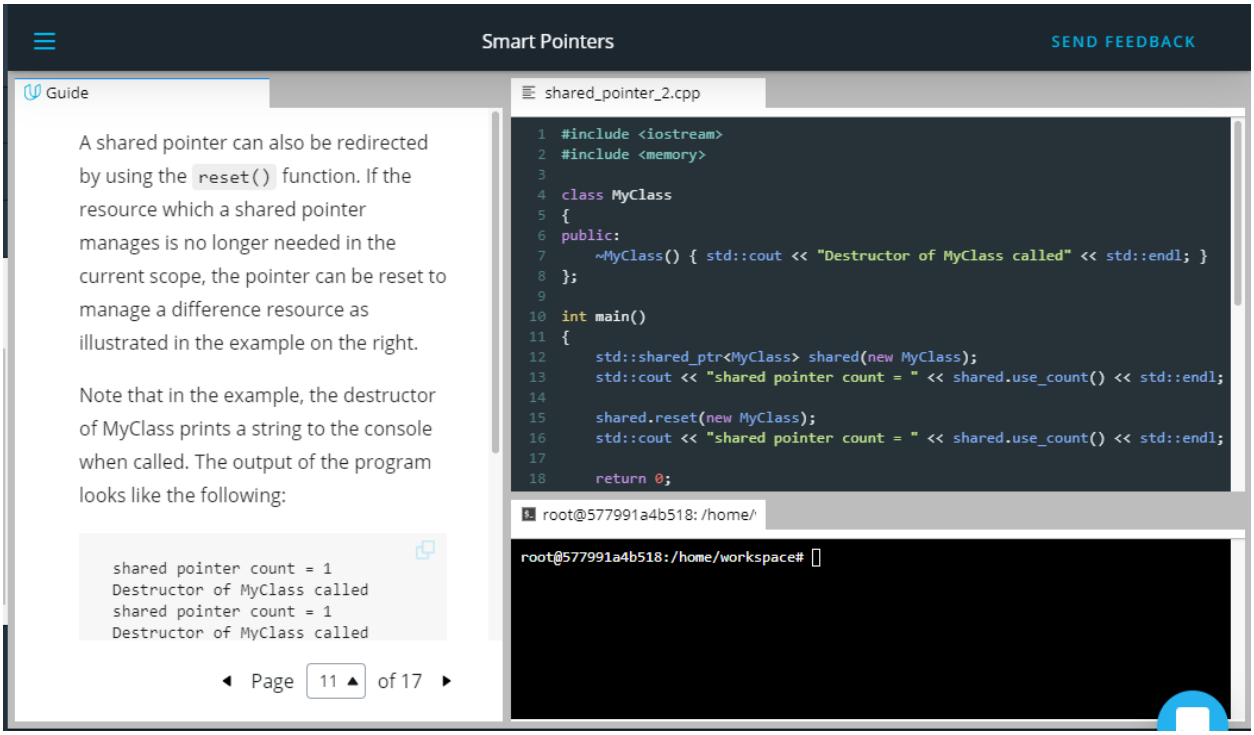


We can see that shared pointers are constructed just as unique pointers are. Also, we can access the internal reference count by using the method use_count(). In the inner block, a second shared pointer shared2 is created and shared1 is assigned to it. In the copy constructor, the internal resource pointer is copied to shared2 and the

resource counter is incremented in both shared1 and shared2. Let us take a look at the output of the code:

```
shared pointer count = 1
shared pointer count = 2
shared pointer count = 1
```

You may have noticed that the lifetime of shared2 is limited to the scope denoted by the enclosing curly brackets. Thus, once this scope is left and shared2 is destroyed, the reference counter in shared1 is decremented by one - which is reflected in the three console outputs given above.



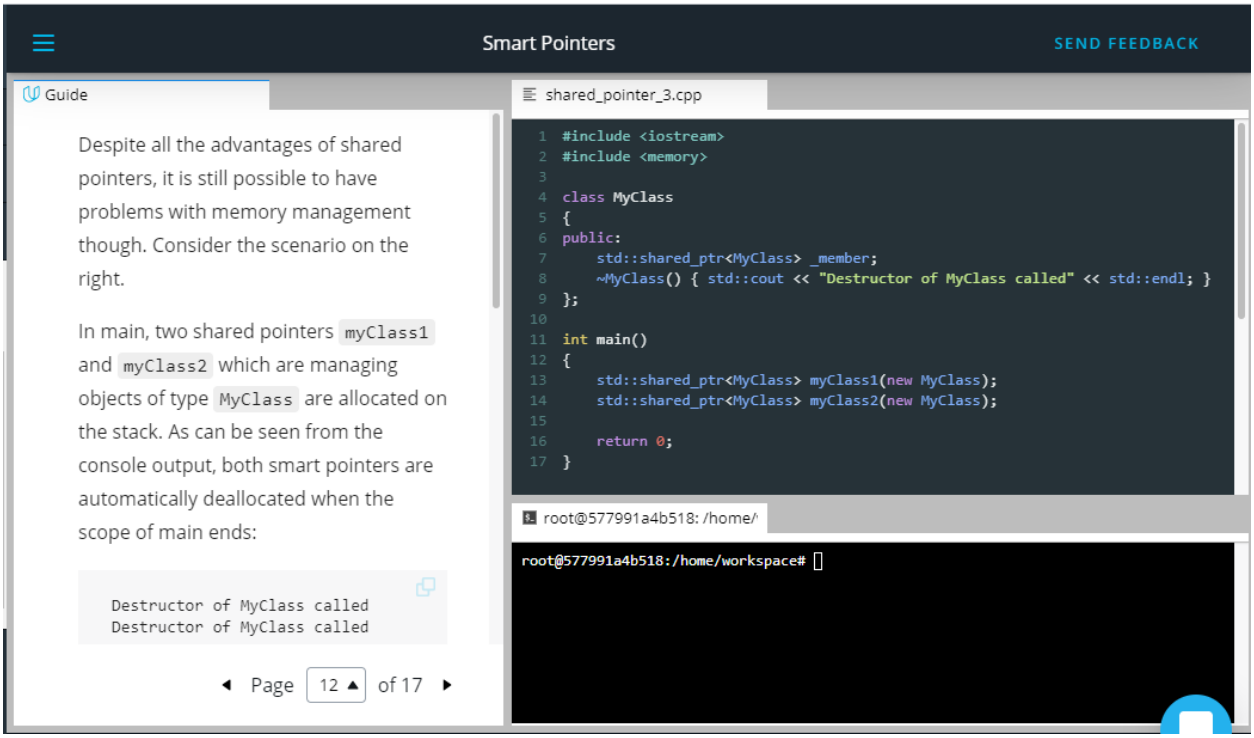
A shared pointer can also be redirected by using the reset() function. If the resource which a shared pointer manages is no longer needed in the current scope, the pointer can be reset to manage a difference resource as illustrated in the example on the right.

Note that in the example, the destructor of MyClass prints a string to the console when called. The output of the program looks like the following:

```
shared pointer count = 1
Destructor of MyClass called
shared pointer count = 1
Destructor of MyClass called
```

After creation, the program prints 1 as the reference count of shared. Then, the reset function is called with a new instance of MyClass as an argument. This causes the destructor of the first MyClass instance to be called, hence the console output. As

can be seen, the reference count of the shared pointer is still at 1. Then, at the end of the program, the destructor of the second MyClass object is called once the path of execution leaves the scope of main.



Despite all the advantages of shared pointers, it is still possible to have problems with memory management though. Consider the scenario on the right.

In main, two shared pointers myClass1 and myClass2 which are managing objects of type MyClass are allocated on the stack. As can be seen from the console output, both smart pointers are automatically deallocated when the scope of main ends:

```
Destructor of MyClass called
Destructor of MyClass called
```

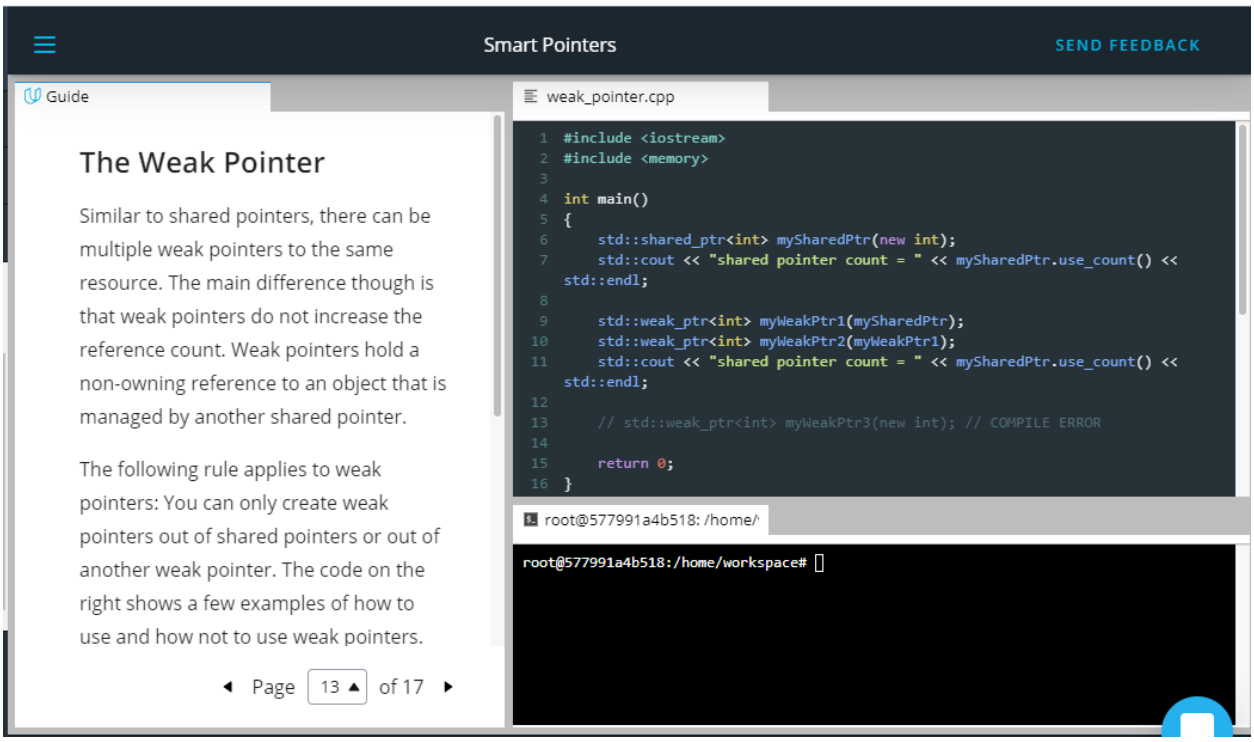
When the following two lines are added to main, the result is quite different:

```
myClass1->_member = myClass2;
myClass2->_member = myClass1;
```

These two lines produce a *circular reference*. When myClass1 goes out of scope at the end of main, its destructor can't clean up memory as there is still a reference count of 1 in the smart pointer, which is caused by the shared pointer _member in myClass2. The same holds true for myClass2, which can not be properly deleted as there is still a shared pointer to it in myClass1. This deadlock situation prevents the destructors from being called and causes a memory leak. When we use *Valgrind* on this program, we get the following summary:

```
==20360== LEAK SUMMARY:
==20360==    definitely lost: 16 bytes in 1 blocks
==20360==    indirectly lost: 80 bytes in 3 blocks
==20360==      possibly lost: 72 bytes in 3 blocks
==20360==    still reachable: 200 bytes in 6 blocks
==20360==         suppressed: 18,985 bytes in 160 blocks
```

As can be seen, the memory leak is clearly visible with 16 bytes being marked as "definitely lost". To prevent such circular references, there is a third smart pointer, which we will look at in the following.



The Weak Pointer

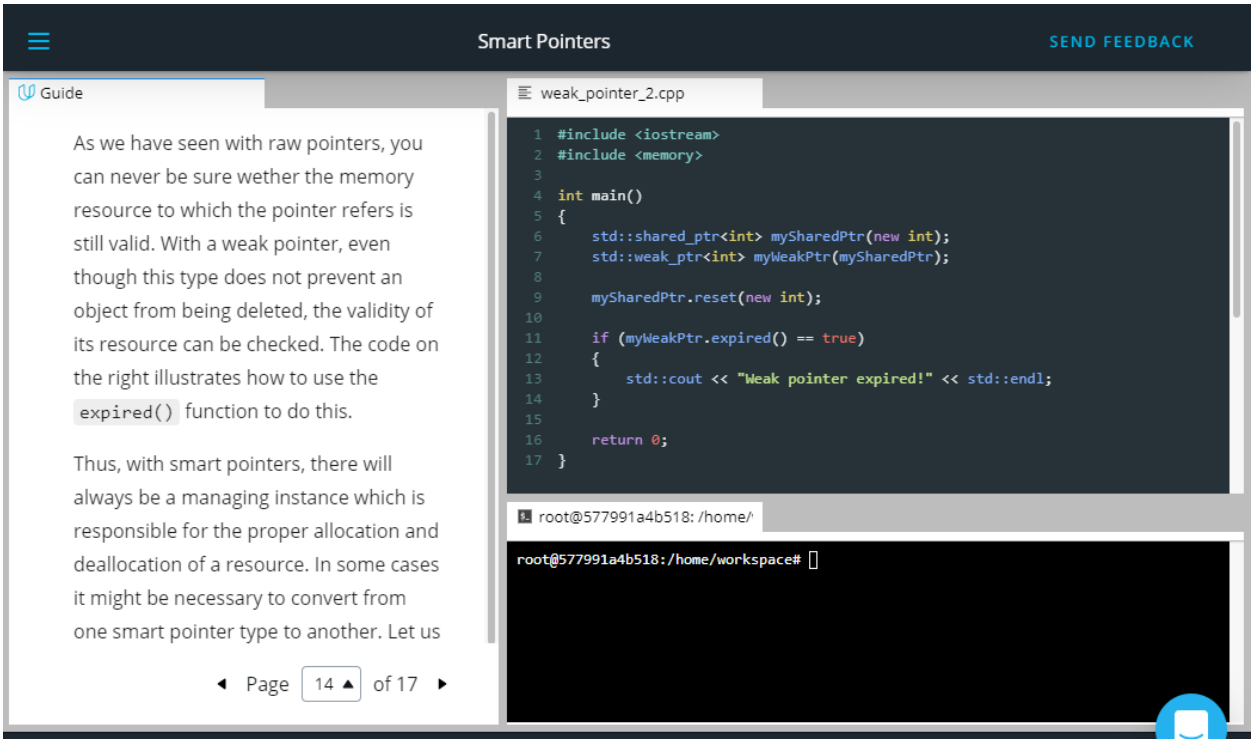
Similar to shared pointers, there can be multiple weak pointers to the same resource. The main difference though is that weak pointers do not increase the reference count. Weak pointers hold a non-owning reference to an object that is managed by another shared pointer.

The following rule applies to weak pointers: You can only create weak pointers out of shared pointers or out of another weak pointer. The code on the right shows a few examples of how to use and how not to use weak pointers.

The output looks as follows:

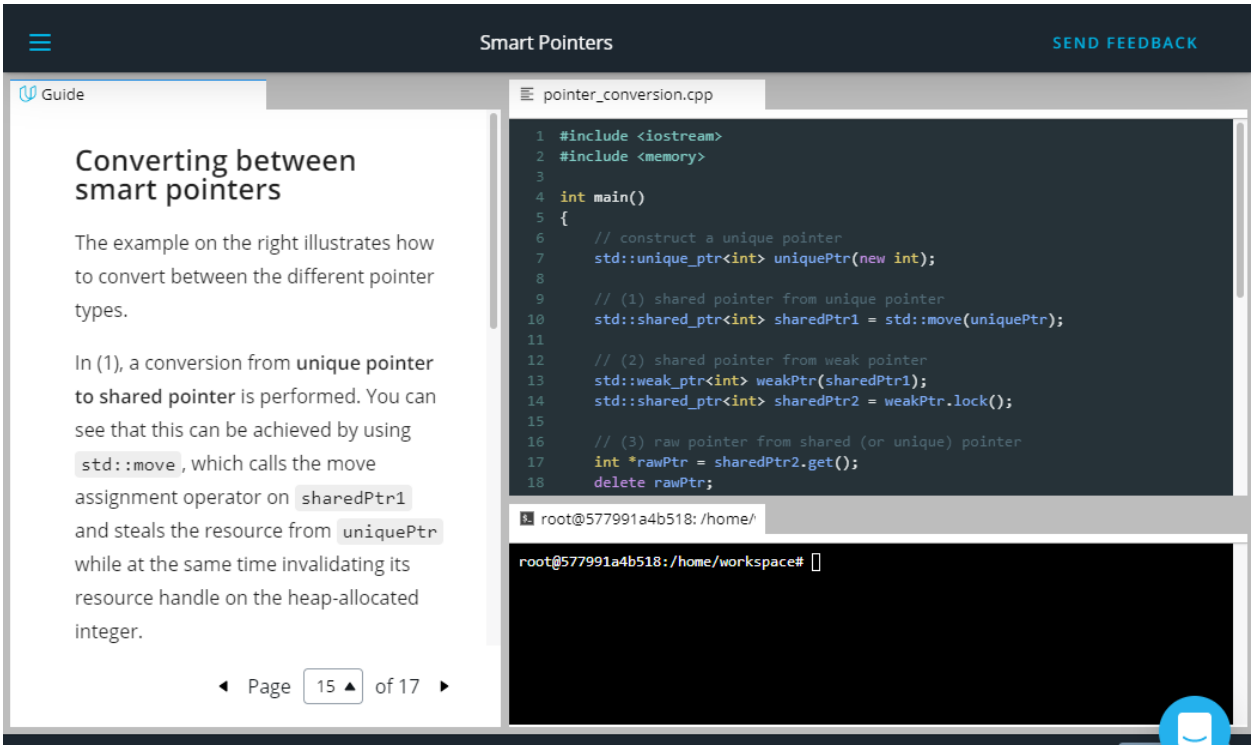
```
shared pointer count = 1
shared pointer count = 1
```

First, a shared pointer to an integer is created with a reference count of 1 after creation. Then, two weak pointers to the integer resource are created, the first directly from the shared pointer and the second indirectly from the first weak pointer. As can be seen from the output, neither of both weak pointers increased the reference count. At the end of main, the attempt to directly create a weak pointer to an integer resource would lead to a compile error.



As we have seen with raw pointers, you can never be sure wether the memory resource to which the pointer refers is still valid. With a weak pointer, even though this type does not prevent an object from being deleted, the validity of its resource can be checked. The code on the right illustrates how to use the `expired()` function to do this.

Thus, with smart pointers, there will always be a managing instance which is responsible for the proper allocation and deallocation of a resource. In some cases it might be necessary to convert from one smart pointer type to another. Let us take a look at the set of possible conversions in the following.

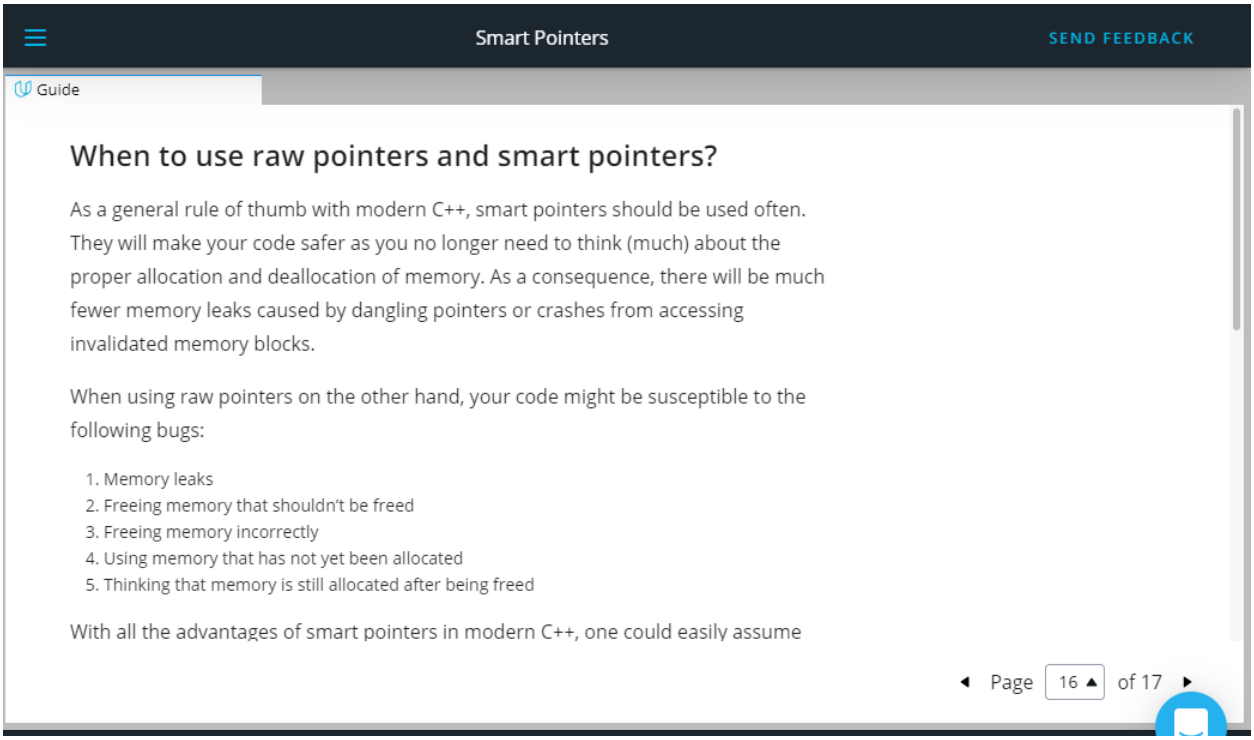


Converting between smart pointers

The example on the right illustrates how to convert between the different pointer types.

In (1), a conversion from **unique pointer to shared pointer** is performed. You can see that this can be achieved by using std::move, which calls the move assignment operator on sharedPtr1 and steals the resource from uniquePtr while at the same time invalidating its resource handle on the heap-allocated integer.
In (2), you can see how to convert **from weak to shared pointer**. Imagine that you have been passed a weak pointer to a memory object which you want to work on. To avoid invalid memory access, you want to make sure that the object will not be deallocated before your work on it has been finished. To do this, you can convert a weak pointer to a shared pointer by calling the lock() function on the weak pointer.
In (3), a **raw pointer is extracted** from a shared pointer. However, this operation does not decrease the reference count within sharedPtr2. This means that calling delete on rawPtr in the last line before main returns will generate a runtime error as a resource is trying to be deleted which is managed by sharedPtr2 and has already been removed. The output of the program when compiled with g++ thus is: malloc: *** error for object 0x1003001f0: pointer being freed was not allocated.

Note that there are **no options for converting away from a shared pointer**. Once you have created a shared pointer, you must stick to it (or a copy of it) for the remainder of your program.



When to use raw pointers and smart pointers?

As a general rule of thumb with modern C++, smart pointers should be used often. They will make your code safer as you no longer need to think (much) about the proper allocation and deallocation of memory. As a consequence, there will be much fewer memory leaks caused by dangling pointers or crashes from accessing invalidated memory blocks.

When using raw pointers on the other hand, your code might be susceptible to the following bugs:

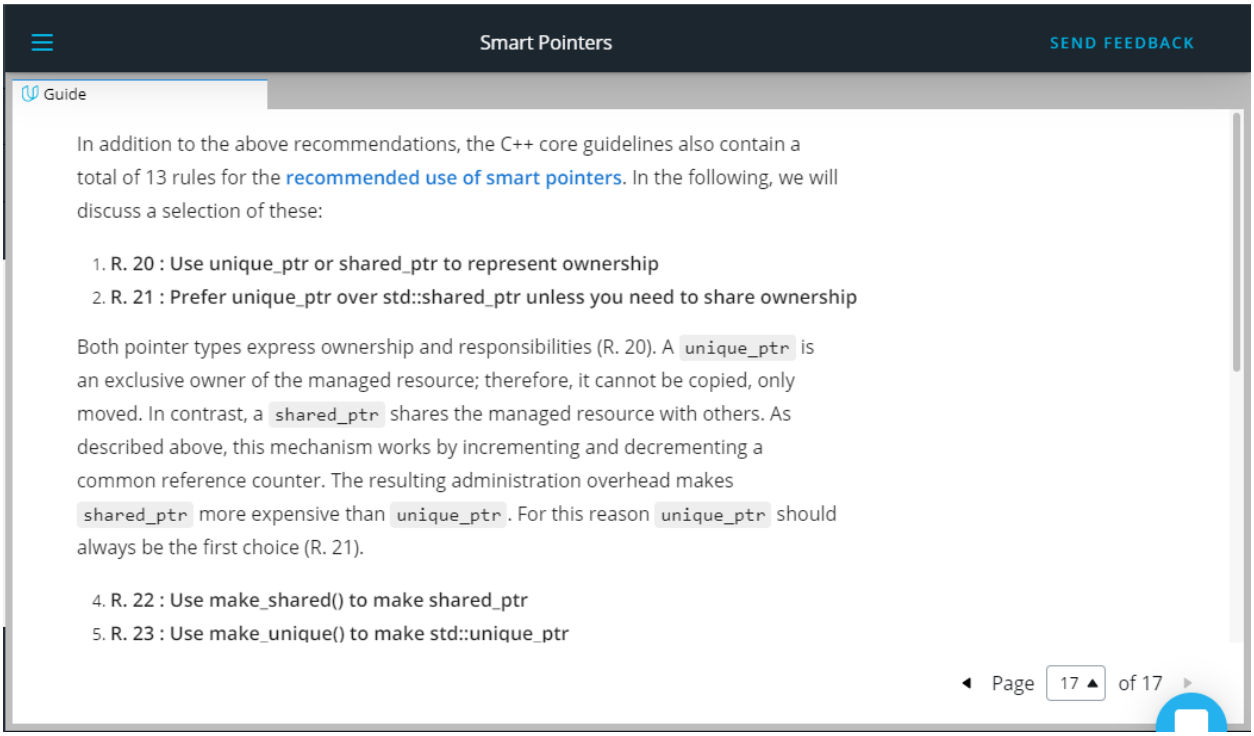
1. Memory leaks
2. Freeing memory that shouldn't be freed
3. Freeing memory incorrectly
4. Using memory that has not yet been allocated
5. Thinking that memory is still allocated after being freed

With all the advantages of smart pointers in modern C++, one could easily assume that it would be best to completely ban the use of new and delete from your code. However, while this is in many cases possible, it is not always advisable as well. Let us take a look at the C++ core guidelines, which has several **rules for explicit memory allocation and deallocation**. In the scope of this course, we will briefly discuss three of them:

1. **R. 10: Avoid malloc and free** While the calls `(MyClass*)malloc(sizeof(MyClass))` and `new MyClass` both allocate a block of memory on the heap in a perfectly valid manner, only `new` will also call the constructor of the class and `free` the destructor. To reduce the risk of undefined behavior, `malloc` and `free` should thus be avoided.
2. **R. 11: Avoid calling new and delete explicitly** Programmers have to make sure that every call of `new` is paired with the appropriate `delete` at the correct position so that no memory leak or invalid memory access occur. The emphasis here lies in the word "explicitly" as opposed to implicitly, such as with smart pointers or containers in the standard template library.
3. **R. 12: Immediately give the result of an explicit resource allocation to a manager object** It is recommended to make use of manager objects for controlling resources such as files, memory or network connections to mitigate the risk of memory leaks. This is the core idea of smart pointers as discussed at length in this section.

Summarizing, raw pointers created with `new` and `delete` allow for a high degree of flexibility and control over the managed memory as we have seen in earlier lessons of this course. To mitigate their proneness to errors, the following additional recommendations can be given:

- A call to `new` should not be located too far away from the corresponding `delete`. It is bad style to stretch you `new` / `delete` pairs throughout your program with references criss-crossing your entire code.
- Calls to `new` and `delete` should always be hidden from third parties so that they must not concern themselves with managing memory manually (which is similar to R. 12).



In addition to the above recommendations, the C++ core guidelines also contain a total of 13 rules for the **recommended use of smart pointers**. In the following, we will discuss a selection of these:

1. **R. 20 : Use unique_ptr or shared_ptr to represent ownership**
2. **R. 21 : Prefer unique_ptr over std::shared_ptr unless you need to share ownership**

Both pointer types express ownership and responsibilities (R. 20). A unique_ptr is an exclusive owner of the managed resource; therefore, it cannot be copied, only moved. In contrast, a shared_ptr shares the managed resource with others. As described above, this mechanism works by incrementing and decrementing a common reference counter. The resulting administration overhead makes shared_ptr more expensive than unique_ptr. For this reason unique_ptr should always be the first choice (R. 21).

4. **R. 22 : Use make_shared() to make shared_ptr**
5. **R. 23 : Use make_unique() to make std::unique_ptr**

The increased management overhead compared to raw pointers becomes in particular true if a shared_ptr is used. Creating a shared_ptr requires (1) the allocation of the resource using new and (2) the allocation and management of the reference counter. Using the factory function make_shared is a one-step operation with lower overhead and should thus always be preferred. (R.22). This also holds for unique_ptr (R.23), although the performance gain in this case is minimal (if existent at all).
But there is an additional reason for using the make_... factory functions: Creating a smart pointer in a single step removes the risk of a memory leak. Imagine a scenario where an exception happens in the constructor of the resource. In such a case, the object would not be handled properly and its destructor would never be called - even if the managing object goes out of scope.
Therefore, make_shared and make_unique should always be preferred. Note that make_unique is only available with compilers that support at least the C++14 standard.

3. **R. 24 : Use weak_ptr to break cycles of shared_ptr**

We have seen that weak pointers provide a way to break a deadlock caused by two owning references which are cyclicly referring to each other. With weak pointers, a resource can be safely deallocated as the reference counter is not increased.

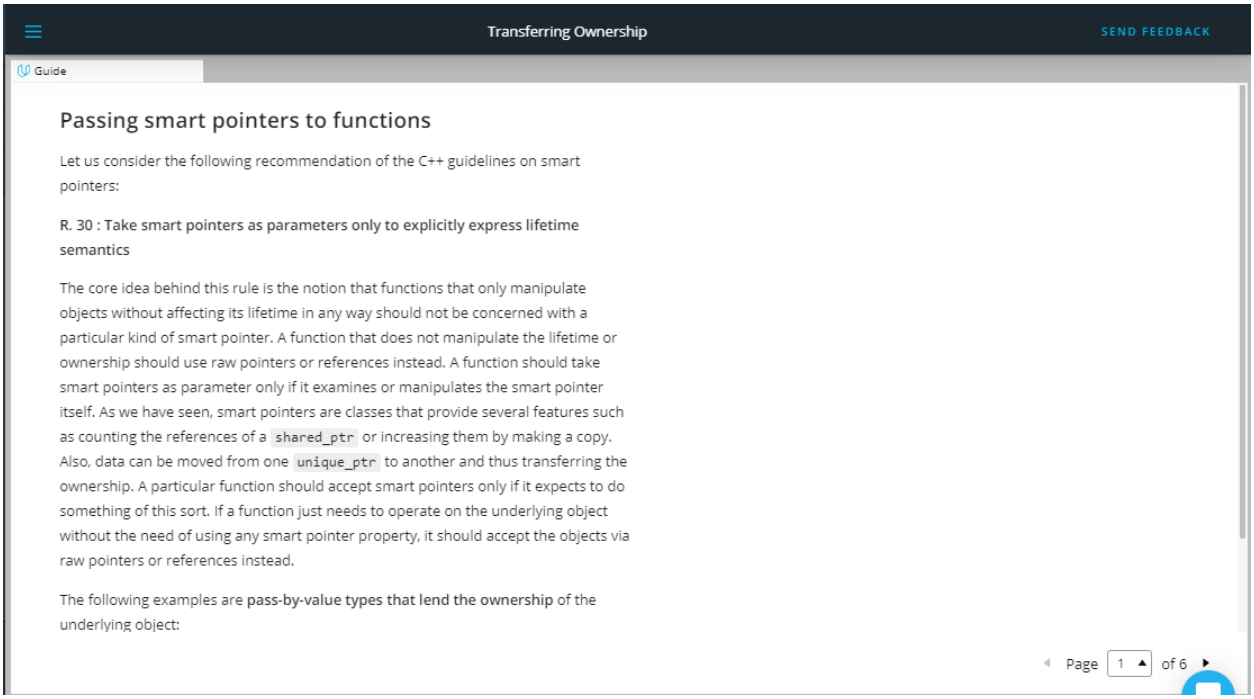
The remaining set of guideline rules referring to smart pointers are mostly concerning the question of how to pass a smart pointer to a function. We will discuss this question in the next concept.

Outro:
https://youtu.be/KsCza62MCSM

4) Bjarne on Smart Pointers
https://youtu.be/4HJ1unZb9l0

5) Transferring Ownership
https://youtu.be/fVnLp5BOvzo
In the previous section, we have taken a look at the three smart pointer types in C++. In addition to smart pointers, you are now also familiar with move semantics, which is of particular importance in this section. In the following, we will discuss how to properly pass and return smart pointers to functions and vice-versa. In modern C++, there are various ways of doing this and in many cases, the method of choice has an impact on both performance and code robustness. The basis of this section

are the C++ core guidelines on smart pointers, some of which we will be examining in the following.



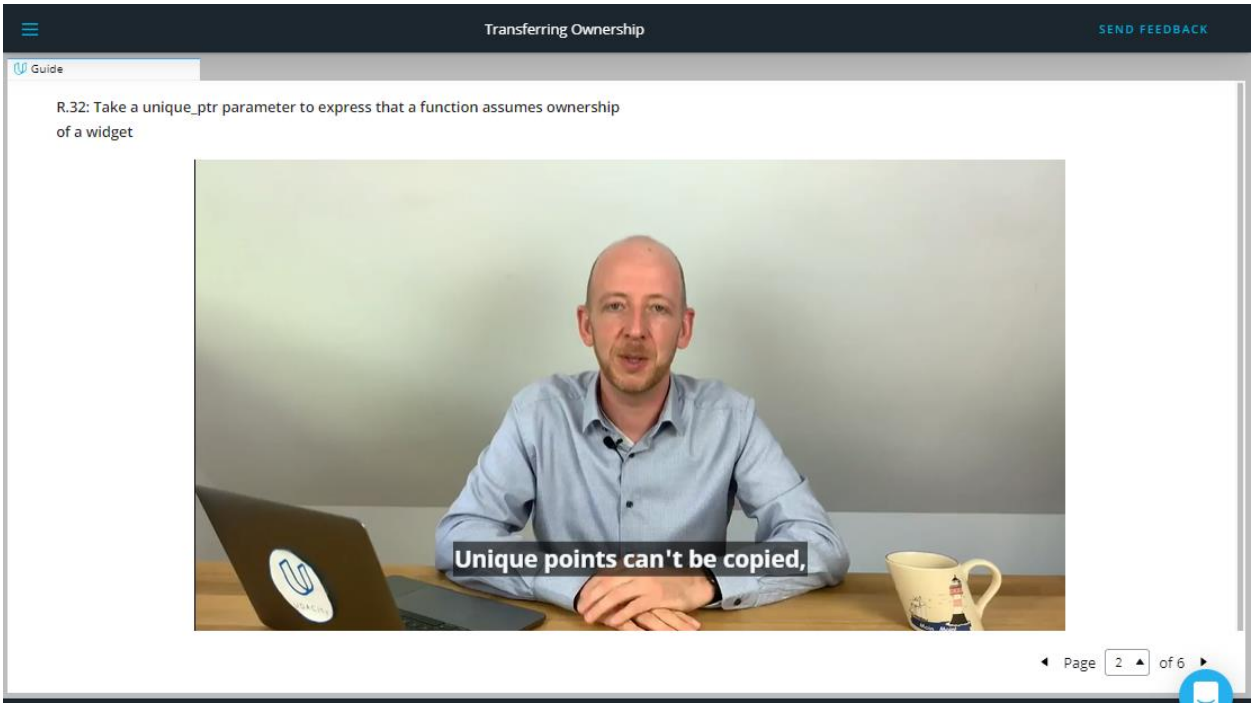
Passing smart pointers to functions

Let us consider the following recommendation of the C++ guidelines on smart pointers:

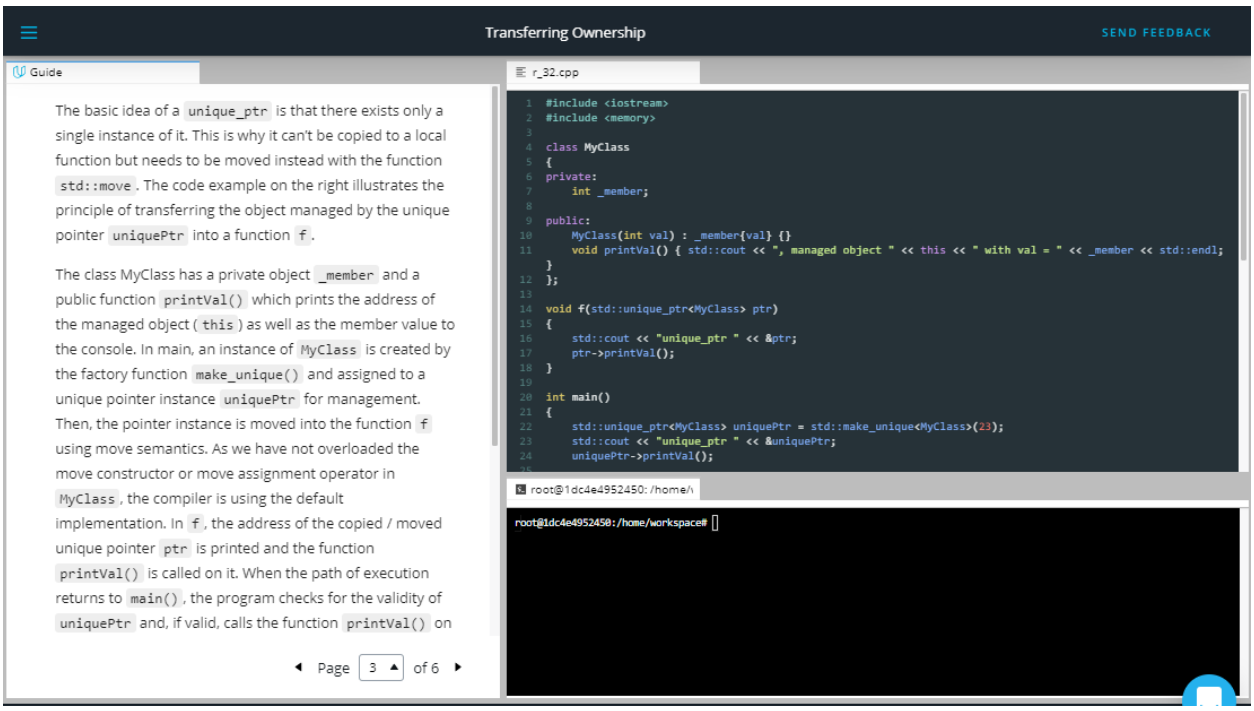
R. 30 : Take smart pointers as parameters only to explicitly express lifetime semantics
The core idea behind this rule is the notion that functions that only manipulate objects without affecting its lifetime in any way should not be concerned with a particular kind of smart pointer. A function that does not manipulate the lifetime or ownership should use raw pointers or references instead. A function should take smart pointers as parameter only if it examines or manipulates the smart pointer itself. As we have seen, smart pointers are classes that provide several features such as counting the references of a shared_ptr or increasing them by making a copy. Also, data can be moved from one unique_ptr to another and thus transferring the ownership. A particular function should accept smart pointers only if it expects to do something of this sort. If a function just needs to operate on the underlying object without the need of using any smart pointer property, it should accept the objects via raw pointers or references instead.
The following examples are **pass-by-value types that lend the ownership** of the underlying object:

1. void f(std::unique_ptr<MyObject> ptr)
2. void f(std::shared_ptr<MyObject> ptr)
3. void f(std::weak_ptr<MyObject> ptr)

Passing smart pointers by value means to lend their ownership to a particular function f. In the above examples 1-3, all pointers are passed by value, i.e. the function f has a private copy of it which it can (and should) modify. Depending on the type of smart pointer, a tailored strategy needs to be used. Before going into details, let us take a look at the underlying rule from the C++ guidelines (where "widget" can be understood as "class").



https://video.udacity-data.com/topher/2019/September/5d865463_nd213-c03-l05-03.2-transferring-ownership-sc/nd213-c03-l05-03.2-transferring-ownership-sc_720p.mp4

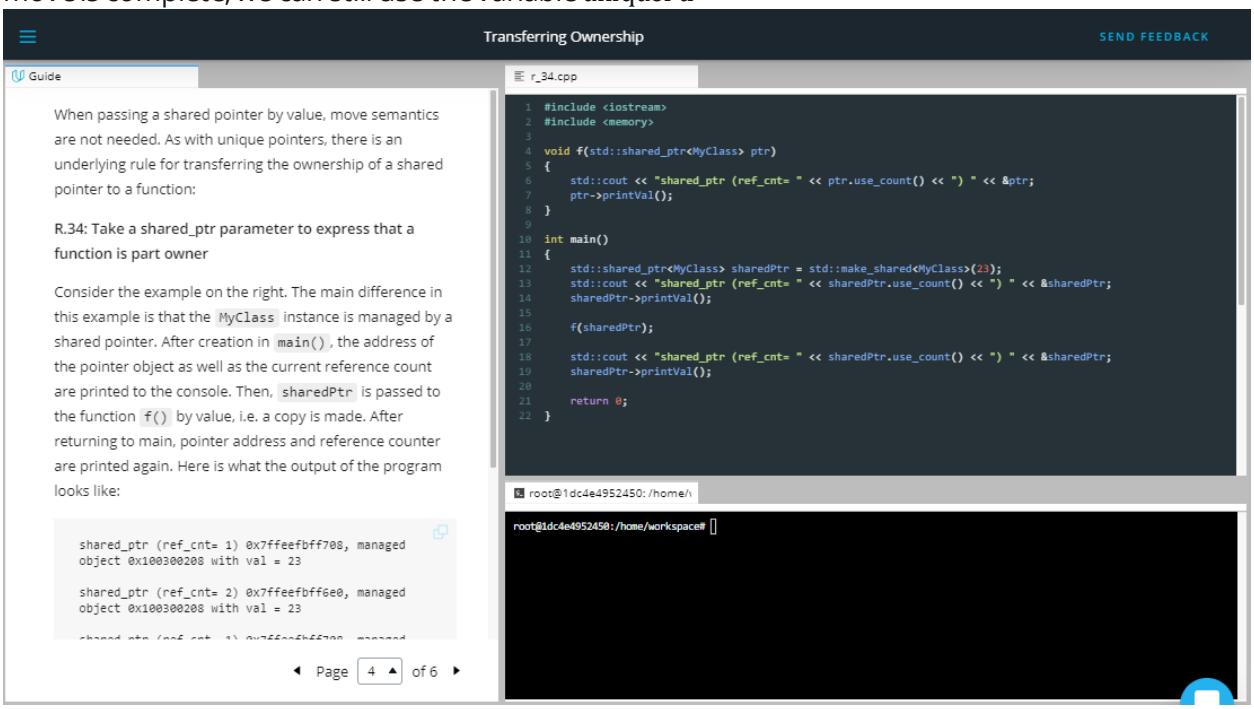


The basic idea of a `unique_ptr` is that there exists only a single instance of it. This is why it can't be copied to a local function but needs to be moved instead with the function `std::move`. The code example on the right illustrates the principle of transferring the object managed by the unique pointer `uniquePtr` into a function `f`.

The class MyClass has a private object _member and a public function printVal() which prints the address of the managed object (this) as well as the member value to the console. In main, an instance of MyClass is created by the factory function make_unique() and assigned to a unique pointer instance uniquePtr for management. Then, the pointer instance is moved into the function f using move semantics. As we have not overloaded the move constructor or move assignment operator in MyClass, the compiler is using the default implementation. In f, the address of the copied / moved unique pointer ptr is printed and the function printVal() is called on it. When the path of execution returns to main(), the program checks for the validity of uniquePtr and, if valid, calls the function printVal() on it again. Here is the console output of the program:

```
unique_ptr 0x7ffeefbff710, managed object 0x100300060 with val = 23

unique_ptr 0x7ffeefbff6f0, managed object 0x100300060 with val = 23
```

The output nicely illustrates the copy / move operation. Note that the address of unique_ptr differs between the two calls while the address of the managed object as well as of the value are identical. This is consistent with the inner workings of the move constructor, which we overloaded in a previous section. The copy-by-value behavior of f() creates a new instance of the unique pointer but then switches the address of the managed MyClass instance from source to destination. After the move is complete, we can still use the variable uniquePtr



When passing a shared pointer by value, move semantics are not needed. As with unique pointers, there is an underlying rule for transferring the ownership of a shared pointer to a function:

R.34: Take a shared_ptr parameter to express that a function is part owner

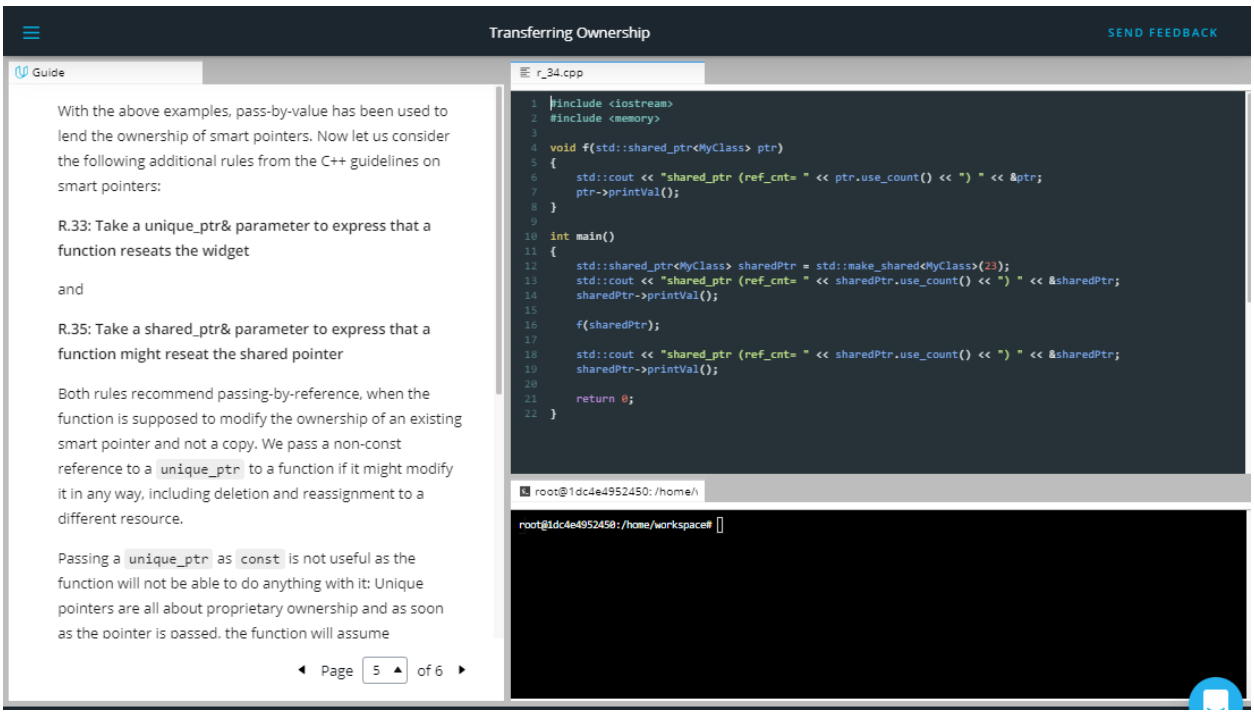
Consider the example on the right. The main difference in this example is that the MyClass instance is managed by a shared pointer. After creation in main(), the address of the pointer object as well as the current reference count are printed to the console. Then, sharedPtr is passed to the function f() by value, i.e. a copy is made. After returning to main, pointer address and reference counter are printed again. Here is what the output of the program looks like:

```
shared_ptr (ref_cnt= 1) 0x7ffeefbff708, managed object 0x100300208 with val = 23
```

```
shared_ptr (ref_cnt= 2) 0x7ffeefbff6e0, managed object 0x100300208 with val = 23

shared_ptr (ref_cnt= 1) 0x7ffeefbff708, managed object 0x100300208 with val = 23
```

Throughout the program, the address of the managed object does not change. When passed to f() , the reference count changes to 2. After the function returns and the local shared_ptr is destroyed, the reference count changes back to 1. In summary, move semantics are usually not needed when using shared pointers. Shared pointers can be passed by value safely and the main thing to remember is that with each pass, the internal reference counter is increased while the managed object stays the same.

Without giving an example here, the weak_ptr can be passed by value as well, just like the shared pointer. The only difference is that the pass does not increase the reference counter.



With the above examples, pass-by-value has been used to lend the ownership of smart pointers. Now let us consider the following additional rules from the C++ guidelines on smart pointers:

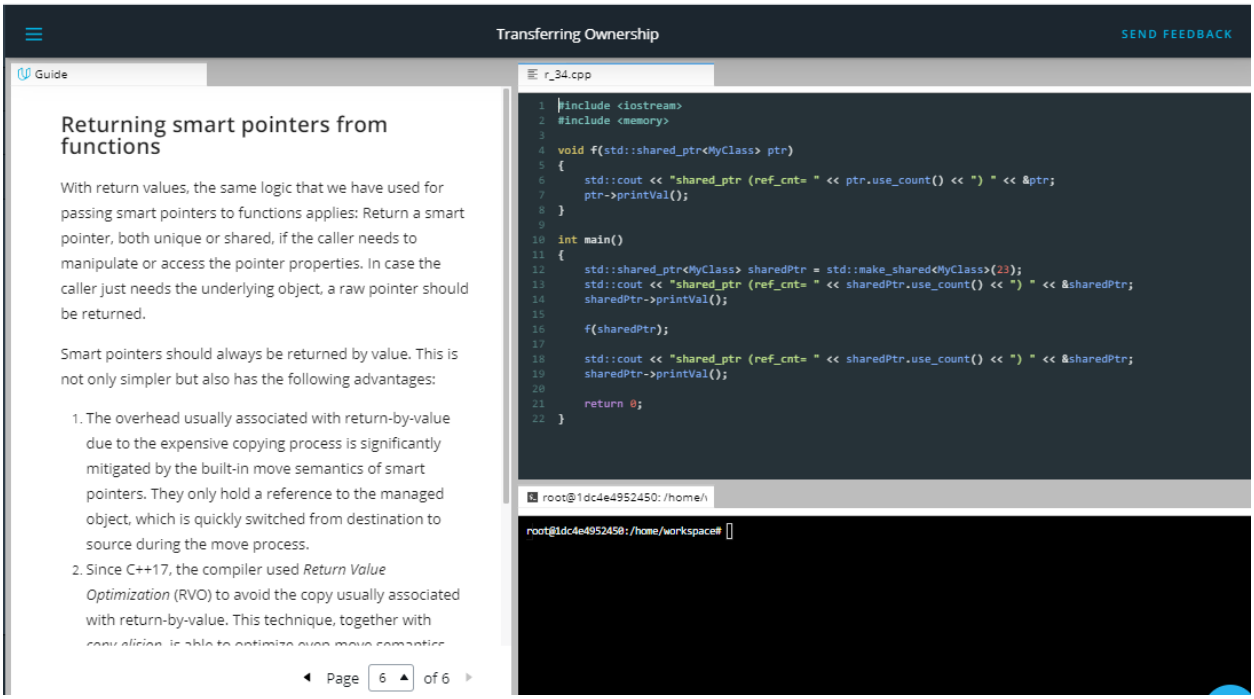
R.33: Take a unique_ptr& parameter to express that a function reseats the widget
and

R.35: Take a shared_ptr& parameter to express that a function might reseat the shared pointer

Both rules recommend passing-by-reference, when the function is supposed to modify the ownership of an existing smart pointer and not a copy. We pass a non-const reference to a unique_ptr to a function if it might modify it in any way, including deletion and reassignment to a different resource.

Passing a unique_ptr as const is not useful as the function will not be able to do anything with it: Unique pointers are all about proprietary ownership and as soon as the pointer is passed, the

function will assume ownership. But without the right to modify the pointer, the options are very limited.
A shared_ptr can either be passed as const or non-const reference. The const should be used when you want to express that the function will only read from the pointer or it might create a local copy and share ownership.
Lastly, we will take a look at **passing raw pointers** and references. The general rule of thumb is that we can use a simple raw pointer (which can be null) or a plain reference (which can ~~not~~ be null), when the function we are passing will only inspect the managed object without modifying the smart pointer. The internal (raw) pointer to the object can be retrieved using the get() member function. Also, by providing access to the raw pointer, you can use the smart pointer to manage memory in your own code and pass the raw pointer to code that does not support smart pointers.
When using raw pointers retrieved from the get() function, you should take special care not to delete them or to create new smart pointers from them. If you did so, the ownership rules applying to the resource would be severely violated. When passing a raw pointer to a function or when returning it (see next section), raw pointers should always be considered as owned by the smart pointer from which the raw reference to the resource has been obtained.



Returning smart pointers from functions

With return values, the same logic that we have used for passing smart pointers to functions applies: Return a smart pointer, both unique or shared, if the caller needs to manipulate or access the pointer properties. In case the caller just needs the underlying object, a raw pointer should be returned.

Smart pointers should always be returned by value. This is not only simpler but also has the following advantages:

1. The overhead usually associated with return-by-value due to the expensive copying process is significantly mitigated by the built-in move semantics of smart pointers. They only hold a reference to the managed object, which is quickly switched from destination to source during the move process.
2. Since C++17, the compiler used *Return Value Optimization* (RVO) to avoid the copy usually associated with return-by-value. This technique, together with *copy-elision*, is able to optimize even move semantics and smart pointers (not in call cases though, they are still an essential part of modern C++).
3. When returning a *shared_ptr* by value, the internal reference counter is guaranteed to be properly incremented. This is not the case when returning by pointer or by reference.

The topic of smart pointers is a complex one. In this course, we have covered many basics and some of the more advanced concepts. However, there are many more aspects to consider and features to use when integrating smart pointers into your code. The full set of smart pointer rules in the C++ guidelines is a good start to dig deeper into one of the most powerful features of modern C++.

Best-Practices for Passing Smart Pointers

This sections contains a condensed summary of when (and when not) to use smart pointers and how to properly pass them between functions. This section is intended as a guide for your future use of this important feature in modern C++ and will hopefully encourage you not to ditch raw pointers altogether but instead to think about where your code could benefit from smart pointers - and when it would most probably not.

The following list contains all the variations (omitting `const`) of passing an object to a function:

```
void f( object* );  //(a)
void f( object& ); //(b)
void f( unique_ptr<object> ); //(c)
void f( unique_ptr<object>& ); //(d)
void f( shared_ptr<object> ); //(e)
void f( shared_ptr<object>& ); //(f)
```

The Preferred Way

The preferred way of to pass object parameters is by using a) or b) :

```
void f( object* );
void f( object& );
```

In doing so, we do not have to worry about the lifetime policy a caller might have implemented. Using a specific smart pointer in a case where we only want to observe an object or manipulate a member might be overly restrictive.

With the non-owning raw pointer `*` or the reference `&` we can observe an object from which we can assume that its lifetime will exceed the lifetime of the function parameter. In concurrency however, this might not be the case, but for linear code we can safely assume this.
To decide wether a * or & is more appropriate, you should think about wether you need to express that there is no object. This can only be done with pointers by passing e.g. `nullptr`. In most other cases, you should use a reference instead.

The Object Sink

The preferred way of passing an object to a function so that the function takes ownership of the object (or „consumes" it) is by using method c) from the above list:

```
void f( unique_ptr<object> );
```

In this case, we are passing a unique pointer by value from caller to function, which then takes ownership of the the pointer and the underlying object. This is only possible using move semantics as there may be only a single reference to the object managed by the unique pointer.

After the object has been passed in this way, the caller will have an invalid unique pointer and the function to which the object now belongs may destroy it or move it somewhere else.

Using `const` with this particular call does not make sense as it models an ownership transfer so the source will be definitely modified.

In And Out Again 1

In some cases, we want to modify a unique pointer (not necessarily the underlying object) and re-use it in the context of the caller. In this case, method d) from the above list might be most suitable:

```
void f( unique_ptr<object>& );
```

Using this call structure, the function states that it might modify the smart pointer, e.g. by redirecting it to another object. It is not recommended to use it for accepting an object only because we should avoid restricting ourselves unnecessarily to a particular object lifetime strategy on the caller side.

Using `const` with this call structure is not recommendable as we would not be able to modify the `unique_ptr` in this case. In case you want to modify the underlying object, use method a) instead.

Sharing Object Ownership

In the last examples, we have looked at strategies involving unique ownership. In this example, we want to express that a function will store and share ownership of an object on the heap. This can be achieved by using method e) from the list above:

```
void f( shared_ptr<object> )
```

In this example, we are making a copy of the shared pointer passed to the function. In doing so, the internal reference counter within all shared pointers referring to the same heap object is incremented by one.

This strategy can be recommended for cases where the function needs to retain a copy of the shared_ptr and thus share ownership of the object. This is of interest when we need access to smart pointer functions such as the reference count or we must make sure that the object to which the shared pointer refers is not prematurely deallocated (which might happen in concurrent programming).

If the local scope of the function is not the final destination, a shared pointer can also be moved, which does not increase the reference count and is thus more effective.

A disadvantage of using a shared_ptr as a function argument is that the function will be limited to using only objects that are managed by shared pointers - which limits flexibility and reusability of the code.

In And Out Again 2

As with unique pointers, the need to modify shared pointers and re-use them in the context of the caller might arise. In this case, method f) might be the right choice:

```
void f( shared_ptr<object>& );
```

This particular way of passing a shared pointer expresses that the function f will modify the pointer itself. As with method e), we will be limiting the usability of the function to cases where the object is managed by a shared_ptr and nothing else.

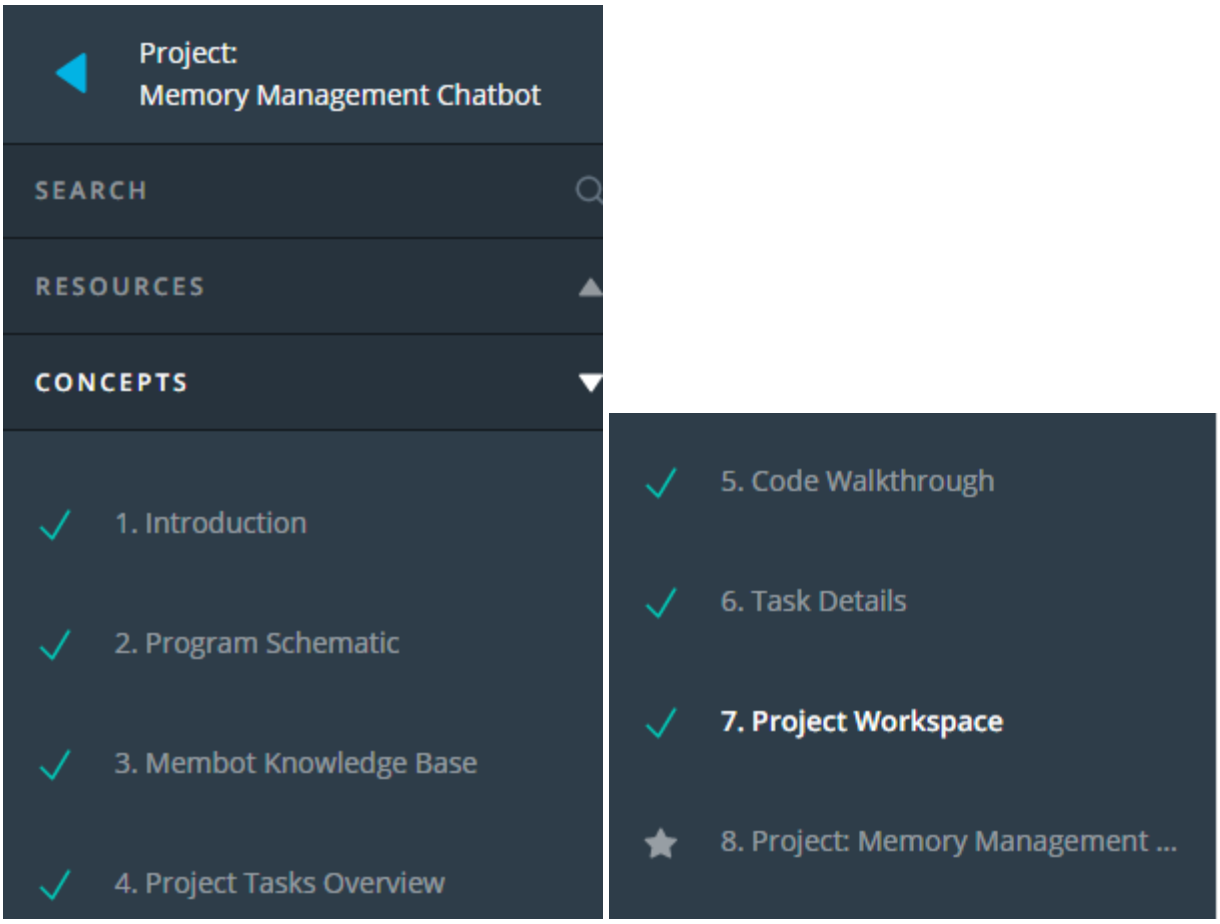
Last Words

The topic of smart pointers is a complex one. In this course, we have covered many basics and some of the more advanced concepts. However, for some cases there are more aspects to consider and features to use when integrating smart pointers into your code. The **full set of smart pointer rules** in the C++ guidelines is a good start to dig deeper into one of the most powerful features of modern C++.

Outro

https://youtu.be/cNqv8RqbuVM

6) Bjarne on Importance of Scope
   https://youtu.be/-k9C4fD1lx4



1) Introduction
https://youtu.be/O37QD8JUPTE

Project Introduction Summary

Purpose : In this project you will analyze and modify an existing ChatBot program, which is able to discuss some memory management topics based on the content of a knowledge base. The program can be executed and works as intended. However, no advanced concepts as discussed in this course have been used. There are no smart pointers, no move semantics and not much thought has been given on ownership and on memory allocation.

Your task : Use the course knowledge to optimize the ChatBot program from a memory management perspective. There is a total of five tasks to be completed. You can find the GitHub repo for the project **here**. The rubric for the project can be found **here**.

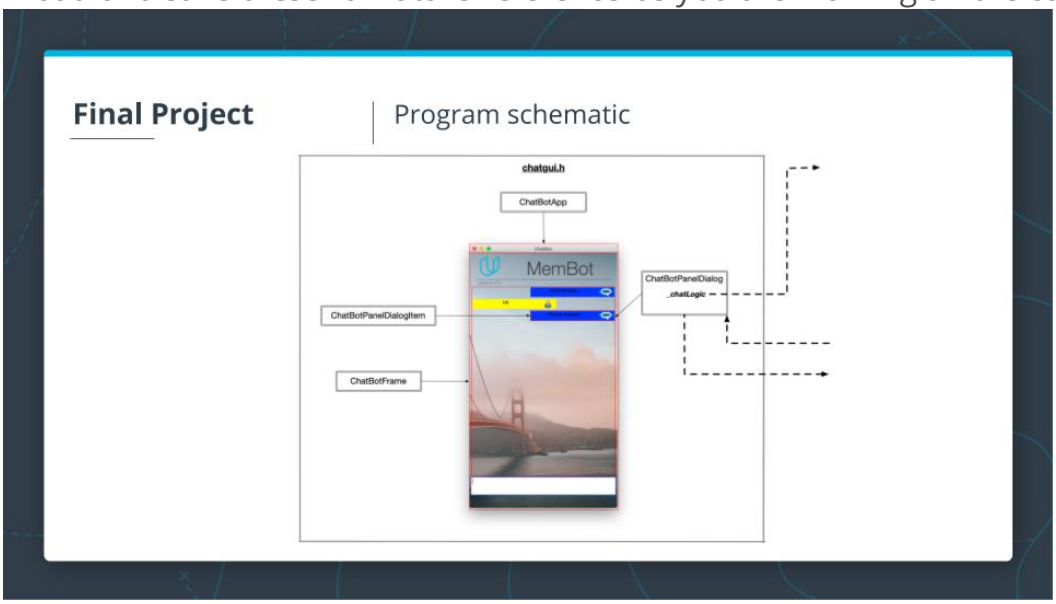
https://github.com/udacity/CppND-Memory-Management-Chatbot

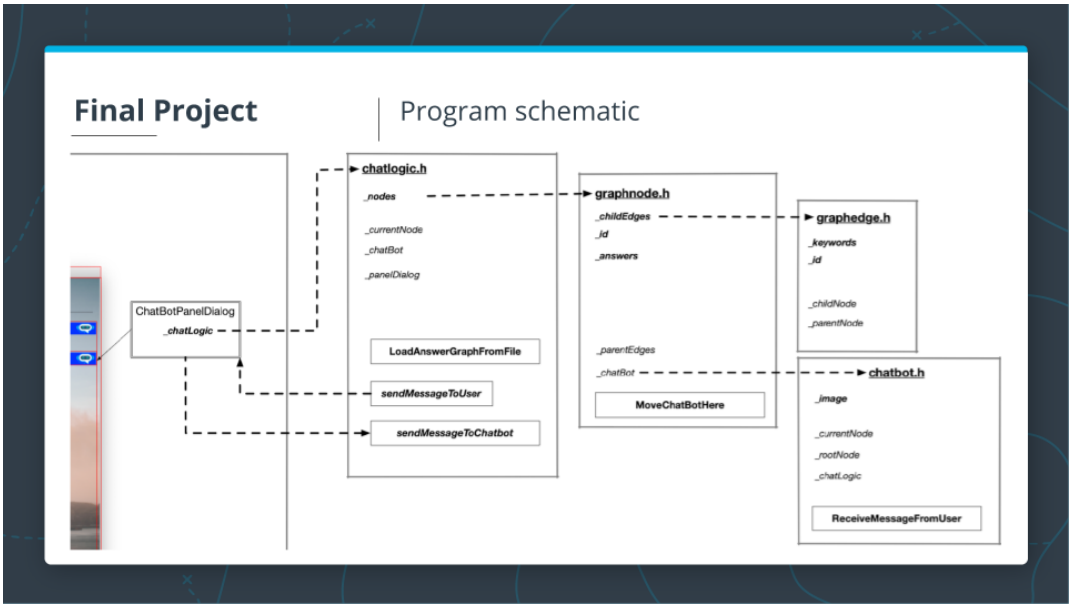
https://review.udacity.com/#!/rubrics/2687/view

2) Program Schematic
https://youtu.be/NIqbxtZ7L6Q
https://youtu.be/XGJIwDYU_ao

Program Schematics

Below are the program schematics from the two screencasts above. You may want to download and save these for future reference as you are working on the code.





3) Membot Knowledge Base
https://youtu.be/dN0_QDto4Cc

4) Project Tasks Overview
https://youtu.be/wY9Lyreewsk
There will be a more detailed discussion of each of these tasks coming up, but first, we will discuss each of the files in the Membot source code.

5) Code Walkthrough

Code Walkthrough

Each of the following screencasts is a detailed overview of the code for one or more files in the project repository. If you haven't seen the repo already, you can find the repo **here**. You may want to download the code to your local machine or open the repo in another browser window to follow along with the screencasts below.

chatgui.h: https://youtu.be/8OKtt-vUqfw
chatgui.cpp: https://youtu.be/RbVb6gl_Ugo
chatlogic.cpp : https://youtu.be/D_PAXGu4GEM
graphnode.h : https://youtu.be/NJKyZVlsXZM
graphnode.cpp: https://youtu.be/lvQYMMm-Dwg
graphedge: https://youtu.be/lXdUrJsxzmc
chatbot.h : https://youtu.be/fWpsTYDSiVU
chatbot.cpp: https://youtu.be/DZt0XlXpc64

6) Task Details

Debug Warm-Up Task
https://youtu.be/ayNGfipZmNg

Task 1 : Exclusive Ownership 1:
https://youtu.be/CggnHc9VJRc

In file `chatgui.h` / `chatgui.cpp`, make _chatLogic an exclusive resource to class `ChatbotPanelDialog` using an appropriate smart pointer. Where required, make changes to the code such that data structures and function parameters reflect the new structure.

Task 2 : The Rule Of Five

https://youtu.be/C82m2EOl1zY

In file `chatbot.h` / `chatbot.cpp`, make changes to the class ChatBot such that it complies with the Rule of Five. Make sure to properly allocate / deallocate memory resources on the heap and also copy member data where it makes sense to you. In each of the methods (e.g. the copy constructor), print a string of the type „ChatBot Copy Constructor“ to the console so that you can see which method is called in later examples.

Task 3 : Exclusive Ownership 2

https://youtu.be/c6OVZXuviCc

In file `chatlogic.h` / `chatlogic.cpp`, adapt the vector _nodes in a way that the instances of `GraphNodes` to which the vector elements refer are exclusively owned by the class `ChatLogic`. Use an appropriate type of smart pointer to achieve this. Where required, make changes to the code such that data structures and function parameters reflect the changes. When passing the `GraphNode` instances to functions, make sure to not transfer ownership and try to contain the changes to class `ChatLogic` where possible.

Task 4 : Moving Smart Pointers

https://youtu.be/kbhdL7MgeIE

In files `chatlogic.h` / `chatlogic.cpp` and `graphnodes.h` / `graphnodes.cpp` change the ownership of all instances of `GraphEdge` in a way such that each instance of `GraphNode` exclusively owns the outgoing `GraphEdges` and holds non-owning references to incoming `GraphEdges`. Use appropriate smart pointers and where required, make changes to the code such that data structures and function parameters reflect the changes. When transferring ownership from class `ChatLogic`, where all instances of `GraphEdge` are created, into instances of `GraphNode`, make sure to use move semantics.

Task 5 : Moving the ChatBot

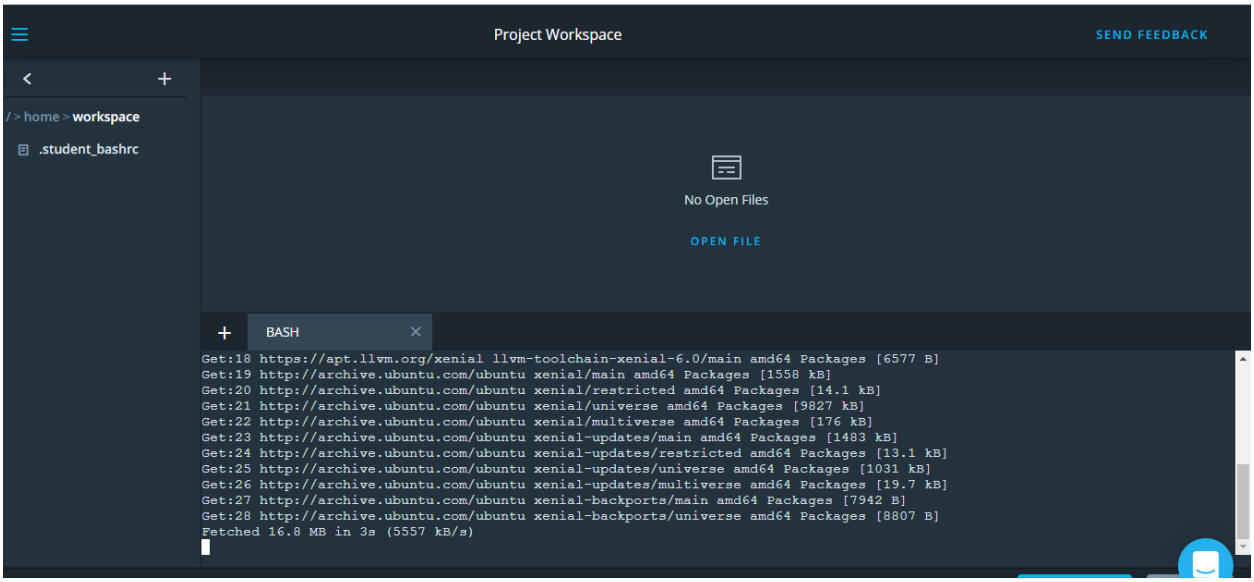
https://youtu.be/amDd3JEpvl8

In file `chatlogic.cpp`, create a local `ChatBot` instance on the stack at the bottom of function `LoadAnswerGraphFromFile`. Then, use move semantics to pass the `ChatBot` instance into the root node. Make sure that `ChatLogic` has no ownership relation to the `ChatBot` instance and thus is no longer responsible for memory allocation and deallocation. Note that the member `_chatBot` remains so it can be used as a communication handle between GUI and `ChatBot` instance. Make all required changes in files `chatlogic.h` / `chatlogic.cpp` and `graphnode.h` / `graphnode.cpp`. When the program is executed, messages on which part of the Rule of Five components of `ChatBot` is called should

be printed to the console. When sending a query to the `ChatBot`, the output should look like the following:

```
ChatBot Constructor
ChatBot Move Constructor
ChatBot Move Assignment Operator
ChatBot Destructor
ChatBot Destructor
```

7) Project Workspace



8) PROJECT: MEMORY MANAGEMENT CHATBOT

Project Submission

DUE DATE

Aug 4

STATUS

Not submitted

Due at: Tue, Aug 4 3:00 pm

You can find the GitHub repo for the project **here**. The rubric for the project can be found **here**. See the previous classroom concepts for additional details on the project.