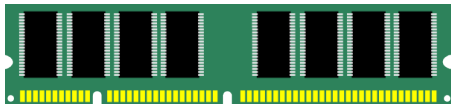


Modern C++ for Computer Vision and Image Processing

Lecture 8: Memory Management

Ignacio Vizzo and Cyrill Stachniss

Working memory or RAM



<http://www.clipartkid.com>

- Working memory has **linear addressing**
- Every byte has an **address** usually presented in hexadecimal form, e.g. `0x7fffb7335fdc`
- Any address can be accessed at random
- **Pointer** is a type to store memory addresses

Pointer

- `<TYPE>*` defines a pointer to type `<TYPE>`
- The pointers **have a type**
- Pointer `<TYPE>*` can point **only** to a variable of type `<TYPE>`
- Uninitialized pointers point to a random address
- Always initialize pointers to an address or a `nullptr`

Example:

```
1 int* a = nullptr;
2 double* b = nullptr;
3 YourType* c = nullptr;
```

Non-owning pointers

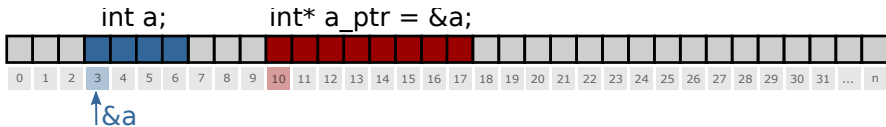
- Memory pointed to by a raw pointer is not removed when pointer goes out of scope
- Pointers can either own memory or not
- Owning memory means being responsible for its cleanup
- **Raw pointers should never own memory**
- We will talk about **smart pointers** that own memory later

Address operator for pointers

- Operator `&` returns the address of the variable in memory
- Return value type is "pointer to value type"
- `sizeof(pointer)` is 8 bytes in 64bit systems

Example:

```
1 int a = 42;  
2 int* a_ptr = &a;
```

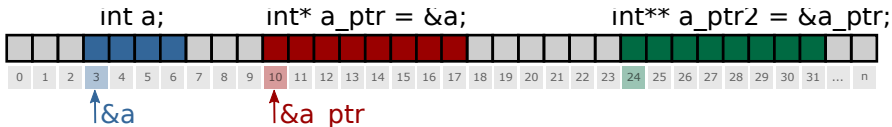


<http://www.cplusplus.com/doc/tutorial/pointers/>

Pointer to pointer

Example:

```
1 int a = 42;  
2 int* a_ptr = &a;  
3 int** a_ptr_ptr = &a_ptr;
```



Pointer dereferencing

- Operator `*` returns the value of the variable to which the pointer points
- Dereferencing of `nullptr`:
Segmentation Fault
- Dereferencing of uninitialized pointer:
Undefined Behavior

Pointer dereferencing

```
1 #include <iostream>
2 using std::cout; using std::endl;
3 int main() {
4     int a = 42;
5     int* a_ptr = &a;
6     int b = *a_ptr;
7     cout << "a = " << a << " b = " << b << endl;
8     *a_ptr = 13;
9     cout << "a = " << a << " b = " << b << endl;
10    return 0;
11 }
```

Output:

```
1 a = 42, b = 42
2 a = 13, b = 42
```


Uninitialized pointer



```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     int* i_ptr; // BAD! Never leave uninitialized!
6     cout << "ptr address: " << i_ptr << endl;
7     cout << "value under ptr: " << *i_ptr << endl;
8     i_ptr = nullptr;
9     cout << "new ptr address: " << i_ptr << endl;
10    cout << "ptr size: " << sizeof(i_ptr) << " bytes";
11    cout << " (" << sizeof(i_ptr) * 8 << "bit) " << endl;
12    return 0;
13 }
```

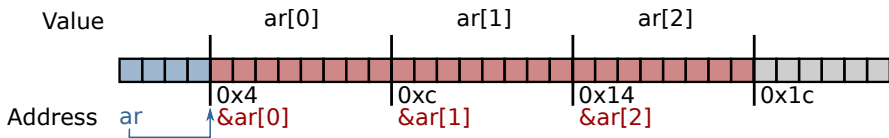
```
1 ptr address: 0x400830
2 value under ptr: -1991643855
3 new ptr address: 0
4 ptr size: 8 bytes (64bit)
```

Important

- Always initialize with a value or a `nullptr`
- Dereferencing a `nullptr` causes a **Segmentation Fault**
- Use `if` to avoid Segmentation Faults

```
1 if(some_ptr) {
2     // only enters if some_ptr != nullptr
3 }
4 if(!some_ptr) {
5     // only enters if some_ptr == nullptr
6 }
```

Arrays in memory and pointers



- Array elements are **continuous in memory**
- Name of an array is an alias to a pointer:

```
1 double ar[3];  
2 double* ar_ptr = ar;  
3 double* ar_ptr = &ar[0];
```

- Get array elements with operator `[]`



Careful! Overflow!

```
1 #include <iostream>
2 int main() {
3     int ar[] = {1, 2, 3};
4     // WARNING! Iterating too far!
5     for (int i = 0; i < 6; i++){
6         std::cout << i << ": value: " << ar[i]
7             << "\t addr:" << &ar[i] << std::endl;
8     }
9     return 0;
10 }
```

```
1 0: value: 1   addr:0x7ffd17deb4e0
2 1: value: 2   addr:0x7ffd17deb4e4
3 2: value: 3   addr:0x7ffd17deb4e8
4 3: value: 0   addr:0x7ffd17deb4ec
5 4: value: 4196992  addr:0x7ffd17deb4f0
6 5: value: 32764  addr:0x7ffd17deb4f4
```

Using pointers for classes

- Pointers can point to objects of custom classes:

```
1 std::vector<int> vector_int;  
2 std::vector<int>* vec_ptr = &vector_int;  
3 MyClass obj;  
4 MyClass* obj_ptr = &obj;
```

- Call object functions from pointer with `->`

```
1 MyClass obj;  
2 obj.MyFunc();  
3 MyClass* obj_ptr = &obj;  
4 obj_ptr->MyFunc();
```

- `obj->Func()` \leftrightarrow `(*obj).Func()`

Pointers are polymorphic

- Pointers are just like references, but have additional useful properties:
 - Can be reassigned
 - Can point to “nothing” (`nullptr`)
 - Can be stored in a vector or an array

■ Use pointers for polymorphism

```
1 Derived derived;  
2 Base* ptr = &derived;
```

- **Example:** for implementing strategy store a pointer to the strategy interface and initialize it with `nullptr` and check if it is set before calling its methods

```
1 struct AbstractShape {
2     virtual void Print() const = 0;
3 };
4 struct Square : public AbstractShape {
5     void Print() const override { cout << "Square\n"; }
6 };
7 struct Triangle : public AbstractShape {
8     void Print() const override { cout << "Triangle\n"; }
9 };
10
11 int main() {
12     std::vector<AbstractShape*> shapes;
13     Square square;
14     Triangle triangle;
15     shapes.push_back(&square);
16     shapes.push_back(&triangle);
17     for (const auto& shape : shapes) {
18         shape->Print();
19     }
20     return 0;
21 }
```

this pointer

- Every object of a class or a struct holds a pointer to itself
- This pointer is called `this`
- Allows the objects to:
 - Return a reference to themselves: `return *this;`
 - Create copies of themselves within a function
 - Explicitly show that a member belongs to the current object: `this->x();`
 - `this` is a C++ keyword

<https://en.cppreference.com/w/cpp/language/this>

Using const with pointers

- Pointers can **point to** a **const** variable:

```
1 // Cannot change value, can reassign pointer.  
2 const MyType* const_var_ptr = &var;  
3 const_var_ptr = &var_other;
```

- Pointers can **be const**:

```
1 // Cannot reassign pointer, can change value.  
2 MyType* const var_const_ptr = &var;  
3 var_const_ptr->a = 10;
```

- Pointers can do both at the same time:

```
1 // Cannot change in any way, read-only.  
2 const MyType* const const_var_const_ptr = &var;
```

- Read from right to left to see which const refers to what

Memory management structures

Working memory is divided into two parts:

Stack and **Heap**



stack

<http://www.freestockphotos.biz>



heap

<https://pixabay.com>

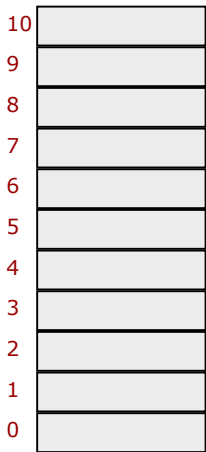
Stack memory



- **Static** memory
- Available for **short term** storage (scope)
- **Small / limited** (8 MB Linux typically)
- Memory allocation is **fast**
- **LIFO** (**L**ast **i**n **F**irst **o**ut) structure
- Items added to top of the stack with **push**
- Items removed from the top with **pop**

Stack memory

stack frame



```
1 #include <stdio.h>
2 int main(int argc, char const* argv[]) {
3     int size = 2;
4     int* ptr = nullptr;
5     {
6         int ar[size];
7         ar[0] = 42;
8         ar[1] = 13;
9         ptr = ar;
10    }
11    for (int i = 0; i < size; ++i) {
12        printf("%d\n", ptr[i]);
13    }
14    return 0;
}
```

command: 2 x pop()

Heap memory



- **Dynamic** memory
- Available for **long** time (program runtime)
- Raw modifications possible with `new` and `delete` (usually encapsulated within a class)
- Allocation is slower than stack allocations



Operators `new` and `new []`

- User controls memory allocation (unsafe)
- Use `new` to allocate data:

```
1 // pointer variable stored on stack
2 int* int_ptr = nullptr;
3 // 'new' returns a pointer to memory in heap
4 int_ptr = new int;
5
6 // also works for arrays
7 float* float_ptr = nullptr;
8 // 'new' returns a pointer to an array on heap
9 float_ptr = new float[number];
```

- `new` returns an address of the variable on the heap
- **Prefer using smart pointers!**



Operators `delete` and `delete[]`

- **Memory is not freed automatically!**
- User must remember to free the memory
- Use `delete` or `delete[]` to free memory:

```
1 int* int_ptr = nullptr;
2 int_ptr = new int;
3 // delete frees memory to which the pointer points
4 delete int_ptr;
5
6 // also works for arrays
7 float* float_ptr = nullptr;
8 float_ptr = new float[number];
9 // make sure to use 'delete[]' for arrays
10 delete[] float_ptr;
```

- **Prefer using smart pointers!**



Example: heap memory

```
1 #include <iostream>
2 using std::cout; using std::endl;
3 int main() {
4     int size = 2; int* ptr = nullptr;
5     {
6         ptr = new int[size];
7         ptr[0] = 42; ptr[1] = 13;
8     } // End of scope does not free heap memory!
9     // Correct access, variables still in memory.
10    for (int i = 0; i < size; ++i) {
11        cout << ptr[i] << endl;
12    }
13    delete[] ptr; // Free memory.
14    for (int i = 0; i < size; ++i) {
15        // Accessing freed memory. UNDEFINED!
16        cout << ptr[i] << endl;
17    }
18    return 0;
19 }
```


Memory leak

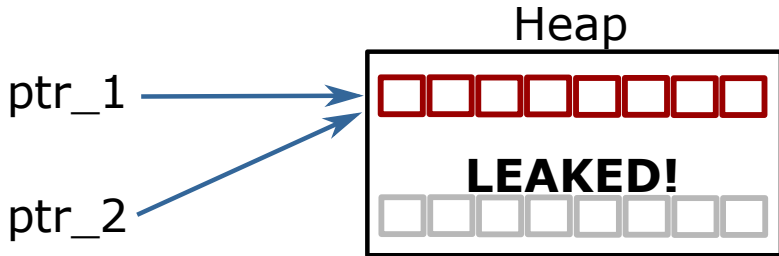
- Can happen when working with Heap memory if we are not careful
- **Memory leak**: memory allocated on Heap access to which has been lost

Memory leak

- Can happen when working with Heap memory if we are not careful
- **Memory leak**: memory allocated on Heap access to which has been lost

Memory leak

- Can happen when working with Heap memory if we are not careful
- **Memory leak**: memory allocated on Heap access to which has been lost



Memory leak (delete)



```
1 int main() {
2     int *ptr_1 = nullptr;
3     int *ptr_2 = nullptr;
4
5     // Allocate memory for two bytes on the heap.
6     ptr_1 = new int;
7     ptr_2 = new int;
8     cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
9
10    // Overwrite ptr_2 and make it point where ptr_1
11    ptr_2 = ptr_1;
12
13    // ptr_2 overwritten, no chance to access the memory.
14    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
15    delete ptr_1;
16    delete ptr_2;
17    return 0;
18 }
```

Error: double free or corruption

```
1 ptr_1: 0x10a3010, ptr_2: 0x10a3070
2 ptr_1: 0x10a3010, ptr_2: 0x10a3010
3 *** Error: double free or corruption (fasttop): 0
   x00000000010a3010 ***
```

- The memory under address `0x10a3070` is **never** freed
- Instead we try to free memory under `0x10a3010` **twice**
- Freeing memory twice is an error

Tools to the rescue

- Standard tools like: `valgrind ./my_program`
- Compiler flags `-fsanitize=address`
- Stackoverflow!



code-fsanitize=address

```
1 =====
2 ==19747==ERROR: AddressSanitizer: attempting double-
   free on 0x602000000010 in thread T0:
3 # ... more stuff
4 0x602000000010 is located 0 bytes inside of 4-byte
5 # ... even more stuff
6 SUMMARY: AddressSanitizer: double-free in operator
   delete(void*, unsigned long)
7 # ... even more more stuff
8 ==19747==ABORTING
```

valgrind output

```
1 HEAP SUMMARY:
2   in use at exit: 4 bytes in 1 blocks
3   total heap usage: 4 allocs, 4 frees, 76,808 bytes
   allocated
4
5 LEAK SUMMARY:
6   definitely lost: 4 bytes in 1 blocks
7   indirectly lost: 0 bytes in 0 blocks
8   possibly lost: 0 bytes in 0 blocks
9   still reachable: 0 bytes in 0 blocks
10  suppressed: 0 bytes in 0 blocks
11 Rerun with --leak-check=full to see details of leaked
   memory
12
13 For counts of detected and suppressed errors, rerun
   with: -v
14 ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
   from 0)
```




Memory leak example

```
1 int main() {
2     double *data = nullptr;
3     size_t size = pow(1024, 3) / 8; // Produce 1GB
4
5     for (int i = 0; i < 4; ++i) {
6         // Allocate memory for the data.
7         data = new double[size];
8         std::fill(data, data + size, 1.23);
9         // Do some important work with the data here.
10        cout << "Iteration: " << i << " done. " << (i + 1)
11            << " GiB has been allocated!" << endl;
12    }
13
14    // This will only free the last allocation!
15    delete[] data;
16    int unused;
17    std::cin >> unused; // Wait for user.
18    return 0;
19 }
```

Memory leak example

- If we run out of memory an `std::bad_alloc` error is thrown
- Be careful running this example, everything might become slow

```
1 # ...
2 Iteration: 19 done. 20 GiB has been allocated!
3 Iteration: 20 done. 21 GiB has been allocated!
4 Iteration: 21 done. 22 GiB has been allocated!
5 Iteration: 22 done. 23 GiB has been allocated!
6 terminate called after throwing an instance of 'std::
   bad_alloc'
7 what(): std::bad_alloc
8 [1]      30561 abort (core dumped)  ./memory_leak_2
```

Dangling pointer

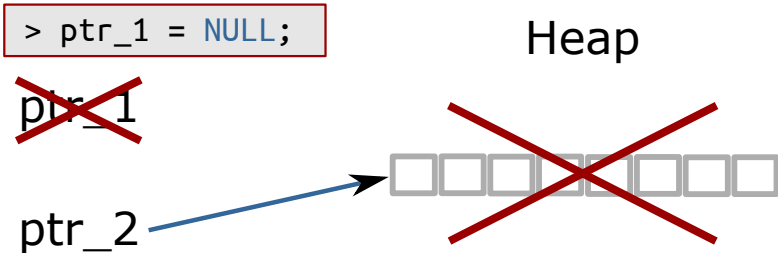
```
1 int* ptr_1 = some_heap_address;  
2 int* ptr_2 = some_heap_address;  
3 delete ptr_1;  
4 ptr_1 = nullptr;  
5 // Cannot use ptr_2 anymore! Behavior undefined!
```

Dangling pointer

```
1 int* ptr_1 = some_heap_address;  
2 int* ptr_2 = some_heap_address;  
3 delete ptr_1;  
4 ptr_1 = nullptr;  
5 // Cannot use ptr_2 anymore! Behavior undefined!
```

Dangling pointer

```
1 int* ptr_1 = some_heap_address;  
2 int* ptr_2 = some_heap_address;  
3 delete ptr_1;  
4 ptr_1 = nullptr;  
5 // Cannot use ptr_2 anymore! Behavior undefined!
```



Dangling pointer

- **Dangling Pointer**: pointer to a freed memory
- Think of it as the opposite of a memory leak
- Dereferencing a dangling pointer causes **undefined behavior**

Dangling pointer example



```
1 #include <iostream>
2 using std::cout; using std::endl;
3 int main() {
4     int size = 5;
5     int *ptr_1 = new int[size];
6     int *ptr_2 = ptr_1; // Point to same data!
7     ptr_1[0] = 100; // Set some data.
8     cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
9     cout << "ptr_2[0]: " << ptr_2[0] << endl;
10    delete[] ptr_1; // Free memory.
11    ptr_1 = nullptr;
12    cout << "1: " << ptr_1 << " 2: " << ptr_2 << endl;
13    // Data under ptr_2 does not exist anymore!
14    cout << "ptr_2[0]: " << ptr_2[0] << endl;
15    return 0;
16 }
```

Even worse when used in functions



```
1 #include <stdio.h>
2 // data processing
3 int* GenerateData(int size);
4 void UseDataForGood(const int* const data, int size);
5 void UseDataForBad(const int* const data, int size);
6 int main() {
7     int size = 10;
8     int* data = GenerateData(size);
9     UseDataForGood(data, size);
10    UseDataForBad(data, size);
11    // Is data pointer valid here? Should we free it?
12    // Should we use 'delete[]' or 'delete'?
13    delete[] data; // ????????????????
14    return 0;
15 }
```


Memory leak or dangling pointer



```
1 void UseDataForGood(const int* const data, int size) {
2     // Process data, do not free. Leave it to caller.
3 }
4 void UseDataForBad(const int* const data, int size) {
5     delete[] data;    // Free memory!
6     data = nullptr;  // Another problem - this does
7                       // nothing!
8 }
```

- **Memory leak** if nobody has freed the memory
- **Dangling Pointer** if somebody has freed the memory in a function

RAII

- **R**esource **A**llocation **I**s **I**nitialization.
- New object → allocate memory
- Remove object → free memory
- Objects **own** their data!

```
1 class MyClass {
2     public:
3         MyClass() { data_ = new SomeOtherClass; }
4         ~MyClass() {
5             delete data_;
6             data_ = nullptr;
7         }
8     private:
9         SomeOtherClass* data_;
10 };
```

- Still cannot copy an object of `MyClass!!!`



CODING
HORROR

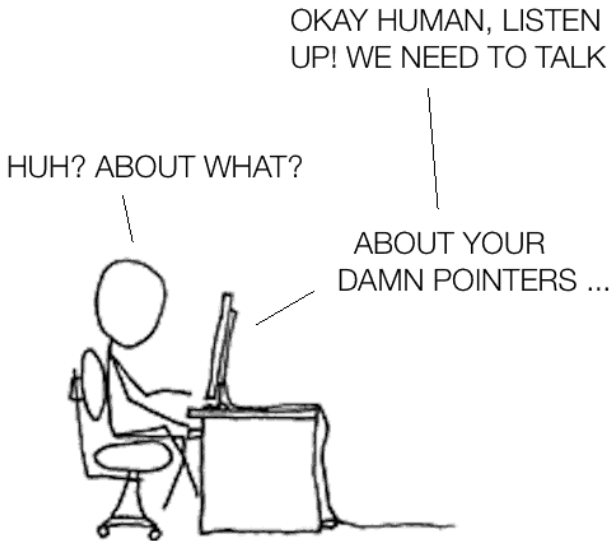
```
1 struct SomeOtherClass {};  
2 class MyClass {  
3     public:  
4         MyClass() { data_ = new SomeOtherClass; }  
5         ~MyClass() {  
6             delete data_;  
7             data_ = nullptr;  
8         }  
9     private:  
10        SomeOtherClass* data_;  
11 };  
12 int main() {  
13     MyClass a;  
14     MyClass b(a);  
15     return 0;  
16 }
```

```
1 *** Error in `raii_example':  
2 double free or corruption: 0x000000000877c20 ***
```

Shallow vs deep copy

- **Shallow copy:** just copy pointers, not data
- **Deep copy:** copy data, create new pointers
- Default copy constructor and assignment operator implement shallow copying
- RAII + shallow copy → **dangling pointer**
- RAII + Rule of All Or Nothing → **correct**
- **Use smart pointers instead!**

Smart pointers



Raw pointers are hard to love

1. Its declaration doesn't indicate whether it points to a single `object` or to an `array`.
2. Its declaration reveals **nothing** about whether you should destroy what it points to when you're done using it, i.e., if the pointer **owns** the thing it points to.
3. If you determine that you should **destroy** what the pointer points to, there's no way to tell how. Should you use `delete`, or is there a different `destruction` mechanism (e.g., a dedicated destruction function the pointer should be passed to)?

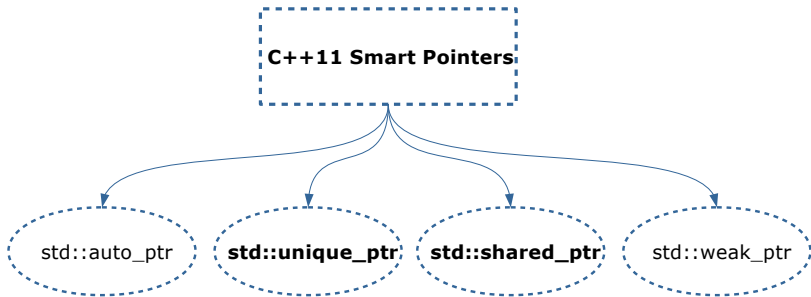
Raw pointers are hard to love

4. If you manage to find out that `delete` is the way to go, Reason 1 means it may not be possible to know whether to use the single-object form ("`delete`") or the `array` form ("`delete []`"). If you use the wrong form, results are **undefined**.
5. There's typically no way to tell if the pointer **dangles**, i.e., points to memory that no longer holds the object the pointer is supposed to point to. Dangling pointers arise when objects are destroyed while pointers still point to them.

Smart pointers

- Smart pointers wrap a raw pointer into a class and manage its lifetime (**RAII**)
- Smart pointers are **all about ownership**
- Always use smart pointers when the pointer should **own heap memory**
- **Only use them with heap memory!**
- Still use raw pointers for non-owning pointers and simple address storing
- `#include <memory>` to use smart pointers

C++11 smart pointers **types**



We will focus on 2 types of smart pointers:

- `std::unique_ptr`
- `std::shared_ptr`

Smart pointers manage memory!

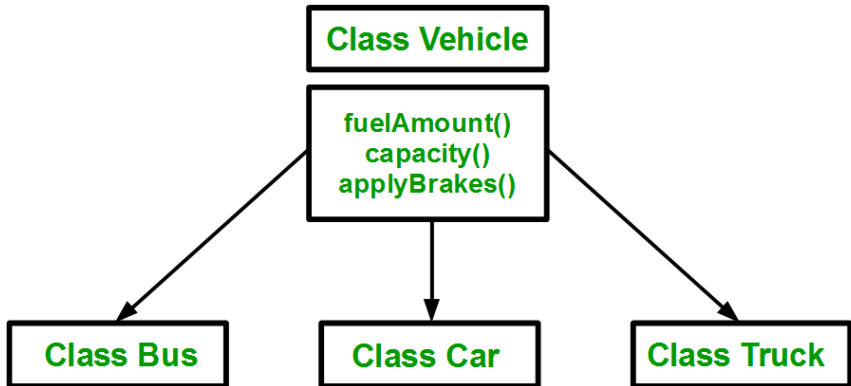
Smart pointers apart from memory allocation behave exactly as raw pointers:

- Can be set to `nullptr`
- Use `*ptr` to dereference `ptr`
- Use `ptr->` to access methods
- Smart pointers are polymorphic

Additional functions of smart pointers:

- `ptr.get()` returns a raw pointer that the smart pointer manages
- `ptr.reset(raw_ptr)` stops using currently managed pointer, freeing its memory if needed, sets `ptr` to `raw_ptr`

std::unique_ptr example



std::unique_ptr example

- Create an `unique_ptr` to a type `Vehicle`

```
1 std::unique_ptr<Vehicle> vehicle_1 =  
2 std::make_unique<Bus>(20, 10, "Volkswagen", "LPM_");  
3  
4 std::unique_ptr<Vehicle> vehicle_2 =  
5 std::make_unique<Car>(4, 60, "Ford", "Sony");
```

- Now you can have fun as we had with `raw pointers`

```
1 // vehicle_x is a pointer, so we can use it as it is  
2 vehicle_1->Print();  
3 vehicle_2->Print();
```

std::unique_ptr example

- `unique_ptr` are **unique**: This means that we can move stuff but **not** copy:

```
1 vehicle_2 = std::move(vehicle_1);
```

- Address of the pointers **before** the move:

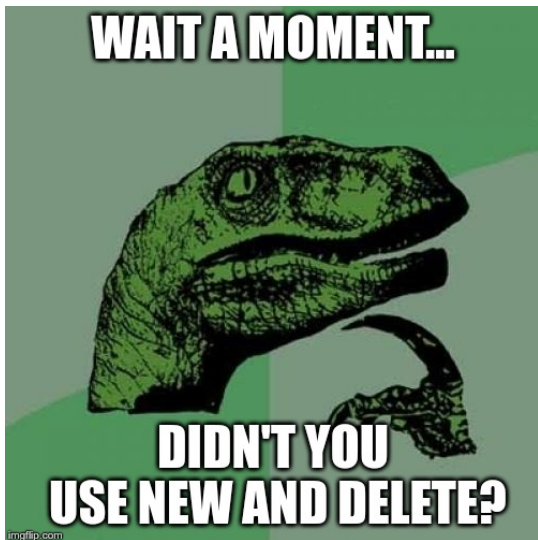
```
1 cout << "vehicle_1 = " << vehicle_1.get() << endl;  
2 cout << "vehicle_2 = " << vehicle_2.get() << endl;
```

```
1 vehicle_1 = 0x56330247ce70  
2 vehicle_2 = 0x56330247cec0
```

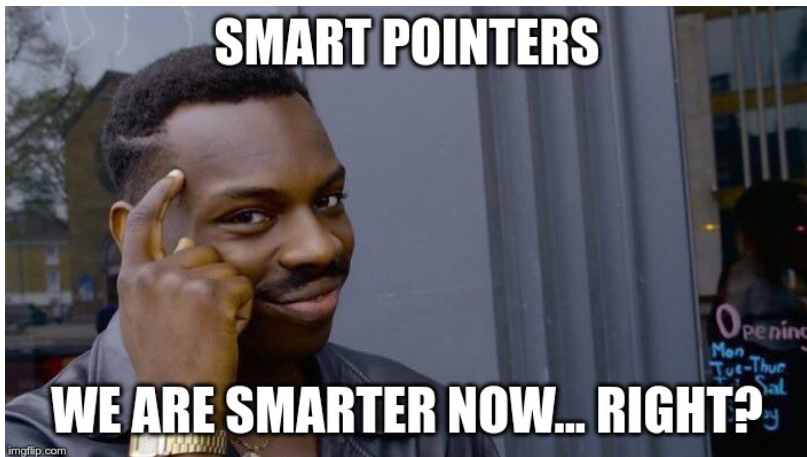
- Address of the pointers **after** the move:

```
1 vehicle_2 = 0x56330247ce70  
2 vehicle_1 = 0
```

std::unique_ptr example



std::unique_ptr example



Unique pointer (`std::unique_ptr`)

- Constructor of a unique pointer takes **ownership** of a provided raw pointer
- **No runtime overhead** over a raw pointer
- Syntax for a unique pointer to type `Type`:

```
1 #include <memory>
2 // Using default constructor Type();
3 auto p = std::unique_ptr<Type>(new Type);
4 // Using constructor Type(<params>);
5 auto p = std::unique_ptr<Type>(new Type(<params>));
```

- From C++14 on:

```
1 // Forwards <params> to constructor of unique_ptr
2 auto p = std::make_unique<Type>(<params>);
```


What makes it “unique”

- Unique pointer **has no copy constructor**
- Cannot be copied, **can be moved**
- Guarantees that memory is **always** owned by a single `std::unique_ptr`
- A non-null `std::unique_ptr` always owns what it points to.
- Moving a `std::unique_ptr` transfers ownership from the source pointer to the destination pointer. (The source pointer is set to `nullptr`.)

Shared pointer (`std::shared_ptr`)

- What if we want to use the same `pointer` for different resources?
- An object accessed via `std::shared_ptrs` has its lifetime managed by those pointers through **shared** ownership.
- No specific `std::shared_ptr` owns the object.
- When the last `std::shared_ptr` pointing to an object stops pointing there, that `std::shared_ptr` destroys the object it points to.

Shared pointer (`std::shared_ptr`)

- Constructed just like a `unique_ptr`
- Can be copied
- Stores a usage counter and a raw pointer
 - Increases usage counter when copied
 - Decreases usage counter when destructed
- Frees memory when counter reaches 0
- Can be initialized from a `unique_ptr`
- Syntax:

```
1 #include <memory>
2 // Using default constructor Type();
3 auto p = std::shared_ptr<Type>(new Type);
4 auto p = std::make_shared<Type>();
5
6 // Using constructor Type(<params>);
7 auto p = std::shared_ptr<Type>(new Type(<params>));
8 auto p = std::make_shared<Type>(<params>);
```

Shared pointer

```
1 class MyClass {
2     public:
3     MyClass() { cout << "I'm alive!\n"; }
4     ~MyClass() { cout << "I'm dead... :(\n"; }
5 };
6
7 int main() {
8     auto a_ptr = std::make_shared<MyClass>();
9     cout << a_ptr.use_count() << endl;
10    {
11        auto b_ptr = a_ptr;
12        cout << a_ptr.use_count() << endl;
13    }
14    cout << "Back to main scope\n";
15    cout << a_ptr.use_count() << endl;
16    return 0;
17 }
```

When to use what?

- Use smart pointers when the pointer **must manage memory**
- By default use `unique_ptr`
- If multiple objects must **share** ownership over something, use a `shared_ptr` to it
- Think of any free standing `new` or `delete` as of a memory leak or a dangling pointer:
 - Don't use `delete`
 - Allocate memory with `make_unique`, `make_shared`
 - Only use `new` in smart pointer constructor if cannot use the functions above

Typical beginner error

```
1 int main() {
2     // Allocate a variable in the stack
3     int a = 42;
4
5     // Create a pointer to that part of the memory
6     int* ptr_to_a = &a;
7
8     // Know stuff about pointers eh?
9     auto a_unique_ptr = std::unique_ptr<int>(ptr_to_a);
10
11    // Same happens with std::shared_ptr.
12    auto a_shared_ptr = std::shared_ptr<int>(ptr_to_a);
13
14    std::cout << "Program terminated correctly!!!\n";
15    return 0;
16 }
```



Typical beginner error

```
1 int* ptr_to_a = &a;
2
3 // Know stuff about pointers eh?
4 auto a_unique_ptr = std::unique_ptr<int>(ptr_to_a);
5
6 // Same happens with std::shared_ptr.
7 auto a_shared_ptr = std::shared_ptr<int>(ptr_to_a);
```

```
1 Program terminated correctly!!!
2 munmap_chunk(): invalid pointer
3 [1] 4455 abort (core dumped) ./wrong_unique
```

- Create a **smart pointer** from a **pointer** to a stack-managed variable
- The variable ends up being owned both by the **smart pointer** and the stack and gets deleted twice → **Error!**

Polymorphism example using smart pointers

```
1 #include <memory>
2 #include <vector>
3 using std::make_unique;
4 using std::unique_ptr;
5 using std::vector;
6
7 int main() {
8     vector<unique_ptr<Rectangle>> shapes;
9     shapes.emplace_back(make_unique<Rectangle>(10, 15));
10    shapes.emplace_back(make_unique<Square>(10));
11
12    for (const auto &shape : shapes) {
13        shape->Print();
14    }
15
16    return 0;
17 }
```


Suggested Video

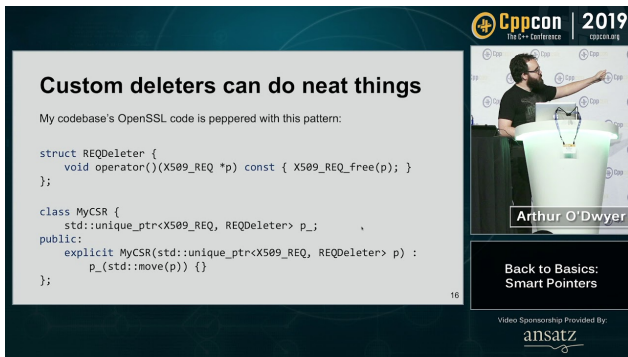
Smart Pointers (short)



<https://youtu.be/UOB7-B2MfwA>

Suggested Video

Smart Pointers (in deep)



The image shows a presentation slide on the left and a video thumbnail on the right. The slide is titled "Custom deleters can do neat things" and contains C++ code for a custom deleter and a class. The video thumbnail shows Arthur O'Dwyer at a podium during a Cppcon 2019 presentation titled "Back to Basics: Smart Pointers".

Custom deleters can do neat things

My codebase's OpenSSL code is peppered with this pattern:

```
struct REQDeleter {
    void operator()(X509_REQ *p) const { X509_REQ_free(p); }
};

class MyCSR {
    std::unique_ptr<X509_REQ, REQDeleter> p_;
public:
    explicit MyCSR(std::unique_ptr<X509_REQ, REQDeleter> p) :
        p_(std::move(p)) {}
};
```

16

Cppcon 2019
The C++ Conference
cppcon.org

Arthur O'Dwyer

**Back to Basics:
Smart Pointers**

Video Sponsorship Provided By:
ansatz

<https://youtu.be/xGDLkt-jBJ4>

References

- **Dynamic Memory Management**

<https://en.cppreference.com/w/cpp/memory>

- **Intro to Smart Pointers**

https://en.cppreference.com/book/intro/smart_pointers

- **Shared Pointers**

https://en.cppreference.com/w/cpp/memory/shared_ptr

- **Unique Pointers**

https://en.cppreference.com/w/cpp/memory/unique_ptr