An introduction to C++

day 6

Sandro Andreotti, Chris Bielow

Bioinformatics Solution Center

Slides: Hannes Hauswedell

Timeframe

day	What	State
Monday	Questions, Recap, Debugging, **Test**	✓
Tuesday	Object-oriented programming 1, Essential operations, Regular types	✓
Wednesday	Memory management, lifetime, pointers	←
Thursday	Object-oriented programming 2, move-semantics, forwarding	
Friday	Type conversions and casts + **Test**	

Object lifetime

new and *delete*

Raw pointers

Smart pointers

Recap from day0

```
#include <iostream>

char c = 'C';   // global scope

int main()
{
    char d = 'D'; // function body scope

    std::cout << c << ' ' << d << '\n';

    { // introduces new local scope
        char e = 'E';
        std::cout << c << ' ' << d << ' ' << e << '\n';
    } // variable e goes "out-of-scope" here

//    std::cout << e << '\n'; // compile-error: not defined
}
```

Lifetime

Disclaimer:

- As with many topics I sometimes don't tell the whole truth and make sensible reductions.
- *"Lifetime"*, *"Scope"* and *"Storage duration"* are closely related (but distinct) terms.
- I won't cover the distinctions here, but there are links at the end of this section that do.

- "Lifetime of an object is equal to or is nested within the lifetime of its storage"
- I henceforth focus on "storage duration".

Storage duration (how)

automatic storage duration:
allocated at beginning of the enclosing code block and deallocated at end

static storage duration:
allocated when the program begins and deallocated when the program ends; only one instance of the object exists

dynamic storage duration:
allocated "manually" by `new` and deleted by `delete`

thread storage duration:
allocated when the thread begins and deallocated when the thread ends; one instance per thread

Storage duration (what)

automatic storage duration:
all local objects, except those declared `static`, `extern` or `thread_local`

static storage duration:
all objects declared at namespace scope (including global namespace), plus those declared with `static` or `extern`

dynamic storage duration:
those objects allocated "manually" by `new`

thread storage duration:
only objects declared `thread_local`; `thread_local` can appear together with `static or extern`.

Storage duration (where)

automatic storage duration:
usually allocated on the *stack*

static storage duration:
usually has separate memory region

dynamic storage duration:
usually allocated on the *heap*

thread storage duration:
usually has separate memory region

Automatic storage duration

```
#include <iostream>

char c = 'C';                              // NOT AUTOMATIC, but static

int main()
{
    char d = 'D';                          // automatic

    std::cout << c << ' ' << d << '\n';

    {
        char e = 'E';                      // automatic
        std::cout << c << ' ' << d << ' ' << e << '\n';
    }                                      // variable e goes "out-of-scope" here

//    std::cout << e << '\n';              // compile-error: not defined
}
```

Static storage duration

```
char c = 'C';                  // global variable (implicit static storage
                               // duration; 'static' keyword means something
                               // else here (internal linkage)!)

void foobar()
{
    static size_t i = 0;       // static "local" variable

    std:: cout << i++ << ' ';
}

struct F
{
    static size_t i = 23;     // static member variable
    size_t j = 42;
};
```

Static keyword (short digression)

```
void foobar()
{
    static size_t i = 0;            // static "local" variable

    std:: cout << i++ << ' ';
}
```

- There is only one `i` in the whole program.
- `i` is allocated on program start and exists until the end of the program.
- Each invocation of `foobar()` prints a larger number.
- The name "`i`" is only visible within this function, not globally.

Static keyword (short digression)

```
struct F
{
    static size_t i = 23;           // static data member
    size_t j = 42;                  // non-static data member
};

std::cout << F::i;

F f;
std::cout << f.i << f.j;
```

- There is only one `i` in the whole program.
- `i` is allocated on program start and exists until the end of the program.
- If `i` is also `public`, it is visible globally under the name `F::i`.
- When an object `f` of type `F` exists, `i` is also accessible as `f.i` (like other members).

Thread storage duration

- "Like `static` duration, but on a per-thread-basis."
- Might be discussed next week, not relevant now.

Dynamic storage duration

- All automatic/static/thread storage is allocated and deallocated without you having to worry about it.
- For **dynamic storage duration**, allocation and de-allocation can happen during any time of program execution.
- It can be governed by **run-time** decisions.

Dynamic storage duration

- All automatic/static/thread storage is allocated and deallocated without you having to worry about it.
- For **dynamic storage duration**, allocation and de-allocation can happen during any time of program execution.
- It can be governed by **run-time** decisions.

In C++ the term "dynamic" usually refers to the opposite of "static". In the case of storage duration this is not fully true, because there is more than "static" and "dynamic", but you should remember that "dynamic" almost always refers to things happening ***at run-time!***

Stack vs. Heap

This is not a class about computer architecture, we won't go into details of the memory.

But you should remember:

1. Dynamic memory needs to be allocated upon first use (and later deallocated), i.e. **dynamic storage might be 'slower'**.
2. On the other hand the heap can grow to be the entire system memory, while the stack is often limited to a few MBs, i.e. **more dynamic storage is possible.**

Being able to allocate user defined types on the stack is one reason C and C++ are faster than other languages (e.g. Java allocates all objects on the heap).

Further reading

- "§1.5 Scope and Lifetime"
  "A Tour of C++", Second Edition, Bjarne Stroustrup
- Storage duration:
  https://en.cppreference.com/w/cpp/language/storage_duration
- Lifetime:
  https://en.cppreference.com/w/cpp/language/lifetime
- Scope:
  https://en.cppreference.com/w/cpp/language/scope

Object lifetime

new and *delete*

Raw pointers

Smart pointers

The fixed array

```
std::array<int64_t, 10> arr{};          // size of array is fixed AT COMPILE-TIME!

for (auto & i : arr)                    // read exactly 10 numbers from input
    std::cin >> i;

for (auto const i : arr)
    std::cout << i << ' ';              // print it back out
```

- This is quite cumbersome and not what you usually want.

The dynamic array

```
std::vector<int64_t> arr{};             // starts with size 0, can change!

for (int64_t buf{}; std::cin >> buf;) // read numbers until Ctrl+D pressed
    arr.push_back(buf);                 // and append to vector

for (auto const i : arr)
    std::cout << i << ' ';              // print it back out
```

- The obvious correct answer is to use a vector, we know that it can grow.
- So it must be doing something fundamentally different; it cannot be known at compile-time how many numbers the user will provide.
- Before we look at what the vector does, we'll look at a simpler example!

The dynamic array

```cpp
size_t s{};
std::cin >> s;                          // ask the user for the size

std::array<int64_t, s> arr{};           // now we know the size when creating arr?

for (auto & i : arr)                    // read exactly s numbers from input
    std::cin >> i;

for (auto const i : arr)
    std::cout << i << ' ';              // print it back out
```

- Q: Does this work?

The dynamic array

```
size_t s{};
std::cin >> s;                          // ask the user for the size

std::array<int64_t, s> arr{};           // now we know the size when creating arr?

for (auto & i : arr)                    // read exactly s numbers from input
    std::cin >> i;

for (auto const i : arr)
    std::cout << i << ' ';              // print it back out
```

- Q: Does this work?
- No, the lifetime of `arr` begins at the beginning of the enclosing block, no matter where it is defined!
- Size is not known at beginning, the value of the template parameter needs to be compile time constant!

The dynamic array

```
size_t s{};
std::cin >> s;                      // ask the user for the size

int64_t * arr = new int64_t[s]{};   // 🧐

for (size_t j = 0; j < s; ++j)      // can't use range-based for loop anymore
    std::cin >> arr[j];

for (size_t j = 0; j < s; ++j)
    std::cout << arr[j] << ' ';     // print it back out

delete[] arr;                       // 🧐
```

- What happens here?

The dynamic array

```
new int64_t[s]
```

- This expression *dynamically* allocates a block memory for `s` variables of type `int64_t`.
- Q: What does it return?

The dynamic array

```
new int64_t[s]
```

- This expression *dynamically* allocates a block memory for `s` variables of type `int64_t`.
- Q: What does it return?
  - A: It returns the *address* of the first variable allocated!

The dynamic array

```
new int64_t[s]
```

- This expression *dynamically* allocates a block memory for `s` variables of type `int64_t`.
- Q: What does it return?
  - A: It returns the *address* of the first variable allocated!
- Q: Why does it not just return the whole array?

The dynamic array

```
new int64_t[s]
```

- This expression *dynamically* allocates a block memory for `s` variables of type `int64_t`.
- Q: What does it return?
  - A: It returns the *address* of the first variable allocated!
- Q: Why does it not just return the whole array?
  - A: Because we wouldn't know in which kind of type to hold the array (becaus types are fixed at compile-time and arrays of different sizes are different types).

The dynamic array

```
new int64_t[s]
```

- This expression *dynamically* allocates a block memory for `s` variables of type `int64_t`.
- Q: What does it return?
  - A: It returns the *address* of the first variable allocated!
- Q: Why does it not just return the whole array?
  - A: Because we wouldn't know in which kind of type to hold the array (because types are fixed at compile-time and arrays of different sizes are different types).
- Q: Ok, so how do we store an *address*?

The dynamic array

```
new int64_t[s]
```

- This expression *dynamically* allocates a block memory for `s` variables of type `int64_t`.
- Q: What does it return?
  - A: It returns the *address* of the first variable allocated!
- Q: Why does it not just return the whole array?
  - A: Because we wouldn't know in which kind of type to hold the array (because types are fixed at compile-time and arrays of different sizes are different types).
- Q: Ok, so how do we store an *address*?
  - A: In a pointer! `int64_t *` is a pointer-to-int64_t, it can hold the address of an `int64_t` variable in memory.

The dynamic array

```
new int64_t                     // allocate space for one int64_t
new int64_t{42}                 // ... and initialise it with '42'

new int64_t[n]                  // allocate space for array of size n
new int64_t[n]{}                // ... and initialise values to 0
```

```
int64_t * i = new int64_t{42};      // pointer to int64_t
int64_t * j = new int64_t[42]{};    // pointer to first element of int64_t[42]

delete i;                           // deallocates single variable
delete[] j;                         // deallocates array
```

When the lifetime of a raw pointer ends, it does nothing with its 'pointee'.

Never forget to delete something allocated with new!

Object lifetime

new and *delete*

Raw pointers

Smart pointers

Working with pointers

```
int64_t i = 42;
int64_t j = 23;
int64_t * i_p = nullptr;            // can be initialised to point nowhere

i_p = &i;                           // change where you point to
//    ^ The "reference" symbol is the "address-of" operator here

*i_p = 7;                           // change value of the pointee
// * as unary prefix operator is "contents-of" operator
// also called "dereferencing" or "indirection"

int64_t * i_p2 = new int64_t{7};

i_p2 = i_p;                         // what happens here?
```

Working with pointers

```
int64_t i = 42;
int64_t j = 23;
int64_t * i_p = nullptr;            // can be initialised to point nowhere

i_p = &i;                           // change where you point to
//    ^ The "reference" symbol is the "address-of" operator here

*i_p = 7;                           // change value of the pointee
// * as unary prefix operator is "contents-of" operator
// also called "dereferencing" or "indirection"

int64_t * i_p2 = new int64_t{7};

i_p2 = i_p;                         // what happens here?
```

`i_p2` now points to `i` and it is no longer possible to delete the memory allocated when creating it! **Memory leak**

Working with pointers

```
int64_t * arr = new int64_t[7]{};   // "holding pointer"

arr[0] = 42;                        // assign to 0th value
std::cout << arr[0];                // prints 0th elem ("42")

int64_t * it = arr;                 // create second pointer to begin
*it = 23;                           // assign through pointer
std::cout << *it;                   // also prints 0th elem ("23")

++it;                               // incrementing pointer moves to next
std::cout << *it;                   // prints 1st element ("0")

it += 3;                            // moves to 4th...
delete[] arr;
```

- Array access syntax as usual (no * required).
- Pointers into arrays work as *iterators*.

Working with (raw) pointers

- can represent **single values or arrays**
- hold **new values** (implied ownership) or **existing values** (similar to references)
- but ownership "not implemented" → need to `delete` / `delete[]` yourself!

Working with (raw) pointers

- can represent **single values or arrays**
- hold **new values** (implied ownership) or **existing values** (similar to references)
- but ownership "not implemented" → need to `delete` / `delete[]` yourself!

- pointer types are *regular types* and pointers are *objects* – in contrast to references.
- assignment/comparison/… relate to stored address, not pointee!
- use "address-of" operator `&` on any object to get address.
- use "contents-of" operator `*` on pointer to access the pointee.
- instead of writing `(*my_obj).foobar()`, you can also write `my_obj->foobar()` (two letters shorter 🤓)
- `this` is a pointer to current object, `*this` returns a reference to current object.

MyVector

```
template <typename T>
class MyVector
{
private:
    T* data{};      // = nullptr
    size_t size{};
public:
    MyVector() = default;
    MyVector(MyVector const &) = default;
    MyVector(MyVector &&) = default;
    MyVector & operator=(MyVector const &) = default;
    MyVector & operator=(MyVector &&) = default;

    MyVector(size_t const s)  { size = s; data = new T[s]{}; }

    ~MyVector()               { delete[] data;  }
```

Allocate on construction and delete on destruction!

MyVector

```
    T* begin() { return data;        }
    T* end()   { return data + size; }
};
```

- As previously mentioned, pointers can work as iterators; `end()` needs to return past-the-end. (Is this safe?)
- This implementation is enough to make this data structure work for our previous example!
- Which other useful member functions did I omit? Exercise!

MyVector

```
size_t s{};
std::cin >> s;                        // ask the user for the size

MyVector<int64_t> arr{s};             // give the size on creation

for (auto & i : arr)                  // ask for s ints from user
    std::cin >> i;

for (auto const i : arr)              // print them back out
    std::cout << i << ' ';
```

- Try this code with the previous definition of `MyVector`.
- Remember to press RETURN after each Input.

MyVector

```
template <typename T>
class MyVector
{
private:
    T* data{};
    size_t size{};
public:
    MyVector() = default;
    MyVector(MyVector const &) = default;
    MyVector(MyVector &&) = default;
    MyVector & operator=(MyVector const &) = default;
    MyVector & operator=(MyVector &&) = default;

    MyVector(size_t const s)  { size = s; data = new T[s]{}; }
    ~MyVector()               { delete[] data;  }
```

Q: What happens when we copy an object of our type?

MyVector

```
MyVector<size_t> v1{7};
MyVector<size_t> v2{9};

v1 = v2;
```

Q: What happens when we copy an object of our type?

MyVector

```
1 h4nn3s@larix /tmp % g++-8 -std=c++17  -Wall -Wextra -Werror -fsanitize=address -g test.cpp
h4nn3s@larix /tmp % ./a.out
=================================================================
==3216==ERROR: AddressSanitizer: attempting double-free on 0x607000000090 in thread T0:
    #0 0x7f48ff7600a0 in operator delete[](void*) ../../../../libsanitizer/asan/asan_new_delete.cc:139
    #1 0x400e37 in MyVector<unsigned long>::~MyVector() /tmp/test.cpp:17
    #2 0x400c34 in main /tmp/test.cpp:24
    #3 0x7f48fea582e0 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x202e0)
    #4 0x400a69 in _start (/tmp/a.out+0x400a69)

0x607000000090 is located 0 bytes inside of 72-byte region [0x607000000090,0x6070000000d8)
freed by thread T0 here:
    #0 0x7f48ff7600a0 in operator delete[](void*) ../../../../libsanitizer/asan/asan_new_delete.cc:139
    #1 0x400e37 in MyVector<unsigned long>::~MyVector() /tmp/test.cpp:17
    #2 0x400c28 in main /tmp/test.cpp:25
    #3 0x7f48fea582e0 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x202e0)

previously allocated by thread T0 here:
    #0 0x7f48ff75f350 in operator new[](unsigned long) ../../../../libsanitizer/asan/asan_new_delete.cc:93
    #1 0x400d6c in MyVector<unsigned long>::MyVector(unsigned long) /tmp/test.cpp:16
    #2 0x400bc8 in main /tmp/test.cpp:25
    #3 0x7f48fea582e0 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x202e0)

SUMMARY: AddressSanitizer: double-free ../../../../libsanitizer/asan/asan_new_delete.cc:139 in operator delete[](void*)
==3216==ABORTING
```

MyVector

```cpp
MyVector<size_t> v1{7};
MyVector<size_t> v2{9};

v1 = v2;
```

Q: What happens when we copy an object of our type?

A: `v1`'s size will be 9 and it's data pointer will point to `v2`'s data. The latter is not really expected, it means `v1`'s data is never deleted and `v2`'s data will be deleted twice when both vectors go out of scope 😱

Furthermore it probably was never intended that they share data, because changing one now changes the other...

TIP: Address-Sanitizers and tools like `valgrind` can help find these errors.

MyVector

```
public:
    MyVector() = default;
    MyVector(MyVector const & rhs) { operator=(rhs); }
    MyVector & operator=(MyVector const & rhs)
    {
        if (this != &rhs) {                         //check self assignment
            size = rhs.size;                        // copy size
            delete[] data; data = new T[size];      // deallocate and reallocate
            for (size_t i = 0; i < size; ++i)
                data[i] = rhs.data[i];              // copy every element
        }
        return *this;
    }
```

- Here we provide a user-defined copy assignment operator to handle deallocation and reallocation, as well as copying of the elements.
- Copy construction is defined as delegating to copy assignment.
- Don't provide move constructor and -assignment for now (not even defaulted).

Pitfalls of (raw) pointers

Using (raw) pointers is a frequent source of bugs, e.g.:

- Memory leaks (forgetting to delete)
- Double delete (deleting something twice)
- Use-after-free (using an object after it was deleted)

Pointers are a legacy from C and many uses of pointers can be easily replaced with references in C++.

For those cases where one needs pointers, C++ has invented *smart pointers* that help avoid some of the typical problems.

Object lifetime

new and *delete*

Raw pointers

Smart pointers

Smart pointers

`std::unique_ptr<T>` / `std::unique_ptr<T[]>`:

- cannot be copied
- deletes its pointee when its own lifetime ends
- for "unique" access to some resource

`std::shared_ptr<T>` / `std::shared_ptr<T[]>`:

- can be copied; copies "know" about each other
- when lifetime of **last copy** ends, it deletes its pointee
- for "shared" access to some resource

Smart pointers

`std::unique_ptr<T>` / `std::unique_ptr<T[]>`:

- cannot be copied
- deletes its pointee when its own lifetime ends
- for "unique" access to some resource

`std::shared_ptr<T>` / `std::shared_ptr<T[]>`:

- can be copied; copies "know" about each other
- when lifetime of **last copy** ends, it deletes its pointee
- for "shared" access to some resource

Q: Should we use one of them for `MyVector`? Which one?

MyVector

```
template <typename T>
class MyVector
{
private:
    size_t size{};
    std::unique_ptr<T[]> data{};
public:
    MyVector() = default;
    ~MyVector() = default;          // can now be defaulted

    MyVector(size_t const s) : size{s}, data{new T[s]{}} {}
```

- What about copying? We know that `std::unique_ptr` cannot be copied.

MyVector

```
public:
    MyVector() = default;
    MyVector(MyVector const & rhs) { operator=(rhs); }
    MyVector & operator=(MyVector const & rhs)
    {
        if (this != &rhs) {                         //check self assignment
            size = rhs.size;                        // copy size
            data.reset(new T[size]);                // deallocate and reallocate
            for (size_t i = 0; i < size; ++i)
                data[i] = rhs.data[i];              // copy every element
        }
        return *this;
    }
```

→ Very similar to before, but we don't need to call delete.

Exercise: You can replace the new call with `std::make_unique`!
Exercise: Do other functions need to be adapted?

Resource Acquisition is initialisation (RAII)

An idiom that describes tying resource management to the lifetime of an object:

- resource acquisition (memory allocation) during construction
- resource release (memory deallocation) upon destruction

Resource Acquisition is initialisation (RAII)

An idiom that describes tying resource management to the lifetime of an object:

- resource acquisition (memory allocation) during construction
- resource release (memory deallocation) upon destruction

Using smart pointers we can tie the lifetime and management of a heap-allocated object – the pointee – to the lifetime of a stack-allocated (and automatically managed) object, the pointer.

This makes it easier to reason about resource management and is safer in many situations.

Summary

Summary

- Objects in C++ can have different lifetimes.
- Objects or arrays *dynamically* allocated with `new` are usually allocated from the *heap* and referenced by a pointer.
- This is slower, but more flexible.

Rules-of-thumb:

- Avoid pointers and dynamic storage if possible. (This is C++, not C!)
  → Prefer references to refer to existing other variables.
- If you dynamically allocate, always use smart pointers (instead of raw pointers), because they clean-up after themselves.
- Only use raw pointers if you refer to pre-existing memory/objects and you need to be able to change where you point to or default-initialise.

Further reading

"A Tour of C++", Second Edition, Bjarne Stroustrup

- "§1.7 Pointers, Arrays and References"
- "§13.2 – §13.2.1 Resource Management"

"R: Resource Management" in the CppCoreGuidelines

- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-resource

On RAII:

- https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

Tasks for the computer lab

Tasks for the computer lab I – MyVector

1. What does `const` mean in the following contexts?

```
size_t       * const ip = new size_t{7};
size_t const *       jp = new size_t{7};
size_t const * const kp = new size_t{7};
```

2. How would you declare/initialise/delete a two-dimensional array of dimensions `n` and `m` (`n` and `m` both not known at compile time)?

3. Can you create reference to pointers and vice versa? What does that mean?

Tasks for the computer lab II – MyVector

- Implement `MyVector` inside a `my_vector.hpp` as discussed in the lecture.
- Also add the member functions `.size()`, `.resize()`, `operator[]` and `.push_back()`.
- Add `const` versions of the member functions where it makes sense.

- Is it efficient to do a resize on every `.push_back()`?
- Instead separate "size" from "capacity" (where capacity is always >= size and is resized in chunks).
- When do you resize by how much? How does this affect the run-time behaviour?

Tasks for the computer lab III – MyVector

- If you haven't done so already, use `std::unique_ptr<>` for data storage.
- Do other member functions need to be adapted? Use `std::make_unique` instead of `new`.
- Build your program with `-fsanitize=address` and verify that it produces no errors (on Linux)

- In some situations no new memory would need to be allocated, which ones? Implement a check.

- We know that shared pointers can be copied, why not use them for `MyVector`?
- Try out! (also explicitly default the rule-of-six const./operators)