C++ basics

Or “what I wish someone told me 10 years ago”

Saska Dönges

After this lecture you should...

- know what undefined behaviour and memory errors are
- be able to start working with C++

AGENDA

1. High level overview (What even is C++)
2. Hello world!
3. Undefined behaviour
4. Memory management
5. Copy and move semantics
6. A bit about CMake

Interrupt to comment or ask questions! You will get points!

WHAT IS C++

C++ is a programming language created in 1979, as an extension to C.

What is C++

C++ is a programming language created in 1979, as an extension to C.

C++ is a procedural, imperative, functional, object-oriented, generic, modular programming language with support for metaprogramming.

What is C++

C++ is a programming language created in 1979, as an extension to C.

~~C++ is a procedural, imperative, functional, object-oriented, generic, modular programming language with support for metaprogramming.~~

C++ is a federation of multiple concepts. C, C with classes, Template C++ and, STL... Among others.

What is C++

C++ is a programming language created in 1979, as an extension to C.

~~C++ is a procedural, imperative, functional, object-oriented, generic, modular programming language with support for metaprogramming.~~

~~C++ is a procedural, object-oriented, functional, generic programming language with support for metaprogramming.~~

C++ is a cluster****. 20 different languages in a trenchcoat pretending to be one well-adjusted language.

What is C++

C++ is a programming language created in 1979, as an extension to C.

~~C++ is a procedural, imperative, functional, object-oriented, generic, modular programming language with support for metaprogramming.~~

~~C++ is a procedural, object-oriented, functional, generic programming language with support for metaprogramming.~~

C++ is a cluster****. 20 different languages in a trenchcoat pretending to be one well-adjusted language.

And probably the most used systems programming language in the world today.

THE (MAIN) PARTS (YOU SHOULD CARE ABOUT)

Any valid C code is valid C++ code. (ish)

C with classes. Instead of just structs in C, we actually have classes with member functions. YAY.

C++ templating. Compile-time generics. Introduces the idea of doing computation at compile time. And the neat but mindboggling art of template metaprogramming. Don't ask me about template metaprogramming.

STL (standard template library). A full complement of ~~fast~~ not too slow data structures and algorithms provided as standard.

HELLO WORLD!

```cpp
#include <stdio.h>
#include <iostream>

int main(int argc, char const *argv[]) {
    std::string hello = "Hello, world!\n";
    puts("Hello, world!");
    std::fputs(hello.c_str(), stdout);
    printf(hello.c_str());
    std::cout << hello << std::flush;
    std::cout.write(hello.c_str(), hello.size());
    return 0;
}
```

HELLO WORLD!

Or simply

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
}
```

Why don’t we even get a warning for not returning enything?

HELLO WORLD!

The main arguments are possibly familiar

```
int main(int argc, char const *argv[]) {
        std::cout << "Hello, " << argv[1] << "!" << std::endl;
}
```

```
~/scratch$ ./hello Exactum
Hello, Exactum!
~/scratch& ./hello
```

What happens without the parameter?

HELLO WORLD!

The main arguments are possibly familiar

```
int main(int argc, char const *argv[]) {
        std::cout << "Hello, " << argv[1] << "!" << std::endl;
}
```

```
~/scratch$ ./hello Exactum
Hello, Exactum!
~/scratch& ./hello
Hello, ~/scratch$
```

Or `Segmentation fault`, depending on stuff and things.

ARRAY INDEX OUT OF BOUNDS

```
int main(int argc, char const *argv[]) {
    int bla[10];
    for (int i = 0; i < 100; i++) {
        std::cout << bla[i]-- << " " << std::flush;
    }
}
```

What is the code doing?
What happens when we compile and run?
Bugs?

ARRAY INDEX OUT OF BOUNDS

```
int main(int argc, char const *argv[]) {
    int bla[10];
    for (int i = 0; i < 100; i++) {
        std::cout << bla[i]-- << " " << std::flush;
    }
}
```

```
-1226429632 32689 0 0 -872412576 21939 -872413072
21939 82839072 32765 -2017372928 2006707886 -872412576
21939  ... 43551 32765 82843582 32765 82843595
32765 *** stack smashing detected ***: <unknown> terminated
Aborted
```

ARRAY INDEX OUT OF BOUNDS

```
int main(int argc, char const *argv[]) {
    int* bla = new int[10]();
    for (int i = 0; i < 100; i++) {
        std::cout << ++bla[i] << " " << std::flush;
    }
}
```

What about now?
What changed?
What will happen?

ARRAY INDEX OUT OF BOUNDS

```
int main(int argc, char const *argv[]) {
    int* bla = new int[10]();
    for (int i = 0; i < 100; i++) {
        std::cout << ++bla[i] << " " << std::flush;
    }
}
```

```
1 1 1 1 1 1 1 1 1 1 1042 1 842276914 825833016 8247 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 ... 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Or `Segmentation fault` if the OS feels like it.

ARRAY INDEX OUT OF BOUNDS

Don’t assume any bounds checking is done in C++.

`stl` containers typically do bounds checking for `at` functions but not for angle bracket access. `std::vector.at()` is safer but slower.

What happens when you attempt to access memory outside of where you should is undefined.

If the memory is ”owned” by you, and you are not smashing the stack the program will not crash even if everything is completely broken.

If you attempt to access memory you are not allowed to, the OS will segfault the program.

VALGRIND TO THE RESCUE

For the second version:

```
~/scratch$ valgrind ./hello
==2799== Command: ./hello
==2799==
1 1 1 1 1 1 1 1 1 1 ==2799== Invalid read of size 4
==2799==    at 0x10898F: main (in /home/saskeli/scratch/hello)
```

Valgrind may not detect stack smashing.

MORE UNDEFINED BEHAVIOUR

Undefined behaviour is any time something happens for which the C++ standard doesn't define a correct behaviour.

A typical example is reading memory that is not initialized. Reading and writing may or may not work as expected, and the process may or may not mangle itself or get killed by the OS.

Most of the time undefined behaviour encountered in the wild is due to memory errors.

Another trap for new players is bit shifting:

```
uint32_t x = 1;
x = x << n;
```

for any $n \geq 32$ is undefined behaviour.

MEMORY MANAGEMENT

Recall

```
int main(int argc, char const *argv[]) {
    int* bla = new int[10]();
    for (int i = 0; i < 10; i++) {
        std::cout << ++bla[i] << " " << std::flush;
    }
}
```

But now fixed

Or is it?

MEMORY MANAGEMENT

```
==2863== Command: ./hello
==2863==
1 1 1 1 1 1 1 1 1 1 ==2863==
==2863== HEAP SUMMARY:
==2863==     in use at exit: 40 bytes in 1 blocks
==2863==   total heap usage: 3 allocs, 2 frees, 73,768 bytes allocated
==2863==
==2863== LEAK SUMMARY:
==2863==    definitely lost: 40 bytes in 1 blocks
==2863==    indirectly lost: 0 bytes in 0 blocks
==2863==      possibly lost: 0 bytes in 0 blocks
==2863==    still reachable: 0 bytes in 0 blocks
==2863==         suppressed: 0 bytes in 0 blocks
==2863== Rerun with --leak-check=full to see details of leaked memory
```

MEMORY MANAGEMENT

Not really important here, since program termination will free up to leaked memory but...

```
int main(int argc, char const *argv[]) {
    int* bla = new int[10]();
    for (int i = 0; i < 10; i++) {
        std::cout << ++bla[i] << " " << std::flush;
    }
    delete(bla);
}
```

Every `new` should have a matching `delete`, and every `malloc`/`calloc` a matching `free`.

MEMORY LEAKS

Especially important when you have classes that allocate memory internally. For example, we could have a neat dynamic array that keeps track of the sum of contained values:

```
template<class dtype>
class SumVector {
   private:
    uint64_t capacity;
    uint64_t size;
    uint64_t sum;
    dtype* data;
```

MEMORY LEAKS (INTERLUDE)

Actually you'd do the following unless you have a very good reason not to.

```
template<class dtype>
class SumVector {
   private:
    uint64_t sum;
    std::vector<dtype> data;
```

And after this the compiler can probably take care of all memory management automatically.

MEMORY LEAKS

It's common for objects to be (de)allocated "under the hood" when doing assignments and passing objects around.

Any time this happens, there is a risk deallocation of an object with reference members leaves orphaned memory that is permanently lost and only deallocated when the process exits.

This leads us conveniently to the next part

COPY AND MOVE SEMANTICS (SOON)

When is memory allocated* for member variables and storage for `vec`?

```
int something_with_a_vector(T& param) {
    /* loads of omitted code here */
    std::vector<int> vec;
    /* loads of code using vec here */
    return x;
}
```

*Perhaps not actually, but for practical purposes

COPY AND MOVE SEMANTICS (SOON)

Memory for local variables is reserved on stack as part of the memory footprint of the function. This includes the local member variables of the vector.

```
std::vector<int>* something_with_a_vector(T& param) {
    /* loads of omitted code here */
    std::vector<int> vec; // Storage space allocated here (or earlier)
    /* loads of code using vec here */
    return &vec;
} //Storage space is actually (typically) deallocated here.
```

Mhen vec goes out of scope the member variables will be outside the stack but the data will still be there. VERY DANGEROUS!

COPIES

Any time you assign a value to a new variable or pass by value to a function, a copy will be made.[†]

```
T thing; // Some heavy object
T other = thing // Copy of thing

void some_function(T thing) // Will create a copy of what is passed in.
```

This can be potentially costly and dangerous.

[†]Or may not actually be... depending on compiler...

COPIES

Any time a copy of a class `T` is created, the copy constructor `T(const T& other)` will be called.

If you have not created one and not forbidden the creation of one one will be provided by the comipiler.

The default copy constructor only copies members naively.

COPIES

So for our neat `SumVector`:

```
int main() {
    SumVector<int> first;
    first.push_back(1).push_back(2).push_back(3);
    SumVector<int> second = first;
    second.pop().push_back(66).push_back(102);
    first.print();
    second.print();
}
```

What happens?

BROKEN COPIES

```
~/scratch$ ./hello
Size = 3
Sum = 6
1 2 66
Size = 4
Sum = 171
1 2 66 102
free(): double free detected in tcache 2
Aborted
~/scratch$
```

FIXED COPIES

To make copies work correctly we need to manually deal with the allocated `data`.

```
SumVector(const SumVector& other)
    : capacity(other.capacity), size(other.size), sum(other.sum) {
    data = (dtype*)malloc(capacity * sizeof(dtype));
    std::memcpy(data, other.data, size * sizeof(dtype));
}
```

Or better yet `SumVector(const SumVector&) = delete;` , so the compiler stops any copying.

MOVE

Move implemented by the `T(T&& other)` constructor, is usually not that problematic. Semantically, gets called when something is passed by value but the original thing can be discarded.

```
std::move(something) // triggers move constructor

func(object()) // Passing a newly constructed object to a function.

T maker() {
    T new_t_thing();
    return new_t_thing;
} // The newly created T object is returned using a move constructor.
```

For the discarded object, a destructor is called.

MOVE PROBLEMS

For our `SumVector`, unless we manually implement a move constructor, we run into issues with move.

After the object is moved, the destructor is called (and if we are not careful) the data storage will be deallocated.

To fix this, just make sure that the object that we moved from, won't be able to deallocate the data:

```
SumVector(SumVector&& other)
    : capacity(other.capacity), size(other.size), sum(other.sum) {
    data = std::exchange(other.data, nullptr);
}
```

To move or not to move

Moving objects can also be disallowed by deleting the move constructor.
(`T(T&&) = delete` .)

But you may want to keep it if you want to pass the objects around or store them in STL containers. `std::vector` for example uses `std::move` when reallocating internal storage.

Generally try to prefer move over copy, and don't do either if you can avoid it.

What is CMake

CMake is the most used build management tool for C++ projects.

I don't like it because the documentation is... bad, and I hadn't found any good tutorials.

Worth knowing about since you will run into it if you keep working with C++. Especially if you want to be compatible with multiple platforms (Windows).

Apparently `https://cliutils.gitlab.io/modern-cmake/` is a good tutorial.

WHY DON'T WE USE CMAKE ON THE COURSE

Make forces us to know more about what is happening.

CSES does not support CMake

I don't know how to use CMake well enough to teach it.