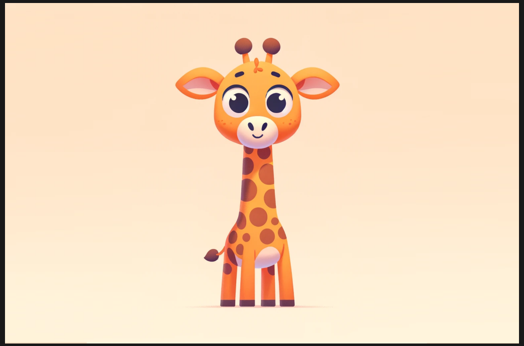
COMP6771

🥑 C++ Basics

Lecture 1.2

Author(s): Hayden Smith



(Download as PDF)

🤏 Basic Types!

Types have defined storage requirements and behaviours. Every variable has a type and this is known at compile time.

🤌 Basic Types!

C++ has a number of standard types you're familiar with from C

```cpp
int main()
{
    // `int` for integers.
    int meaning_of_life = 42;

    // `double` for rational numbers.
    double six_feet_in_metres = 1.8288;

    // `char` for single characters.
    char letter = 'C';

    (void)meaning_of_life;
    (void)six_feet_in_metres;
    (void)letter;
}
```

basic1.cpp

🤏 Basic Types!

As well as many more types

```cpp
#include <string>

int main()
{
    // `string` for text.
    std::string course_code = std::string("COMP6771");

    // `bool` for truth
    bool is_cxx = true;
    bool is_danish = false;

    (void)is_cxx;
    (void)is_danish;
}
```

basic2.cpp

💻 Direct On Hardware

Remember that C++, like C, and unlike Java/Python/Javascript, runs directly on hardware and not through a virtual layer.

That means that some types may differ in properties depending on the system.

💻 Direct On Hardware

C++ actually has useful libraries to help determine this.

```cpp
#include <iostream>
#include <limits>

int main()
{
    std::cout << std::numeric_limits<int>::max() << "\n";
    std::cout << std::numeric_limits<int>::min() << "\n";
    std::cout << std::numeric_limits<double>::max() << "\n";
    std::cout << std::numeric_limits<double>::min() << "\n";
}
```

system-specific.cpp

🧠 `Auto` Keyword

A powerful feature of C++ is the `auto` keyword that allows the compiler to statically infer the type of a variable based on what is being assigned to it on the RHS.

```cpp
int main()
{
    auto i = 0; // i is an int
    auto j = 8.5; // j is a double
    auto k = false; // k is a bool
    (void)i;
    (void)j;
    (void)k;
}
```

auto.cpp

⛔ `const` Keyword

- The const keyword specifies that a value cannot be modified
- Everything should be const unless you know it will be modified
- The course will focus on const-correctness as a major topic

```cpp
int main()
{
    // `int` for integers.
    auto const meaning_of_life = 42;

    // `double` for rational numbers.
    auto const six_feet_in_metres = 1.8288;

    (void)meaning_of_life;
    (void)six_feet_in_metres;

    // meaning_of_life++; // COMPILE ERROR HERE
}
```

const1.cpp

⛔ `const` Keyword

- You can put the const on the left or the right side of the type
- In 6771 the convention is to put it on the right side of the type

```cpp
int main()
{
    auto const meaning_of_life1 = 42; // good
    const auto meaning_of_life2 = 42; // bad
    (void)meaning_of_life1;
    (void)meaning_of_life2;
}
```

const2.cpp

⛔ `const` Keyword

But why?

- Clearer code (you can know a function won't try and modify something just by reading the signature)
- Immutable objects are easier to reason about
- The compiler may be able to make certain optimisations
- Immutable objects are much easier to use in multithreading situations

🦜 Expressions

In computer science, an expression is a combination of values and functions that are interpreted by the compiler to produce a new value.

We will explore some basic expressions in C++

🦜 Expressions

Integers

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{
    auto const x = 10;
    auto const y = 173;

    auto const sum = 183;
    CHECK(x + y == sum);

    auto const difference = 163;
    CHECK(y - x == difference);
    CHECK(x - y == -difference);

    auto const product = 1730;
    CHECK(x * y == product);

    auto const quotient = 17;
    CHECK(y / x == quotient);

    auto const remainder = 3;
    CHECK(y % x == remainder);
}
```

expression-integral.cpp

🦜 Expressions

Floating Points

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{
    auto const x = 15.63;
    auto const y = 1.23;

    auto const sum = 16.86;
    CHECK(x + y == sum);

    auto const difference = 14.4;
    CHECK(x - y == difference);
    CHECK(y - x == -difference);

    auto const product = 19.2249;
    CHECK(x * y == product);

    auto const expected = 12.7073170732;
    auto const actual = x / y;
    auto const acceptable_delta = 0.0000001;
    CHECK(std::abs(expected - actual) < acceptable_delta);
}
```

expression-floating.cpp

🦜 Expressions

Strings

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{

    auto const expr = std::string("Hello, expressions!");
    auto const cxx = std::string("Hello, C++!");

    CHECK(expr != cxx);
    CHECK(expr.front() == cxx[0]);

    auto expr2 = expr;

    // Abort TEST_CASE if expression is false
    REQUIRE(expr == expr2);
}
```

expression-string.cpp

Expressions

Booleans

```cpp
#include <catch2/catch.hpp>

auto const is_comp6771 = true;
auto const is_about_cxx = true;
auto const is_about_german = false;

TEST_CASE()
{
    CHECK((is_comp6771 and is_about_cxx));
    CHECK((is_about_german or is_about_cxx));
    CHECK(not is_about_german);
}

// You can use classic && or || as well
```

expression-boolean.cpp

🚄 C++ Has Value Semantics

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{
    auto const hello = std::string("Hello!");
    auto hello2 = hello;

    // Abort TEST_CASE if expression is false
    REQUIRE(hello == hello2);

    hello2.append("2");
    REQUIRE(hello != hello2);

    CHECK(hello.back() == '!');
    CHECK(hello2.back() == '2');
}
```

value-semantics.cpp

♻️ Type Conversion

In C++ we are able to convert types implicitly or explicitly. We will cover this later in the course in more detail.

♻️ Type Conversion

Implicit promoting conversions

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{
    auto const i = 0;
    {
        auto d = 0.0;
        REQUIRE(d == 0.0);

        d = i; // Silent conversion from int to double
        CHECK(d == 42.0);
        CHECK(d != 41);
    }
}
```

convert-implicit.cpp

♻️ Type Conversion

Explicit promoting conversions

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{
    auto const i = 0;
    {
        // Preferred over implicit, since your intention is clear
        auto const d = static_cast<double>(i);
        CHECK(d == 42.0);
        CHECK(d != 41);
    }
}
```

convert-explicit.cpp

🛠️ Functions

C++ has functions just like other languages. We will explore some together.

🛠️ Functions

Different types of functions

```cpp
#include <catch2/catch.hpp>

bool is_about_cxx()
{ // nullary functions (no parameters)
    return true;
}

int square(int const x)
{ // unary functions (one parameter)
    return x * x;
}

int area(int const width, int const length)
{ // binary functions (two parameters)
    return width * length;
}

TEST_CASE()
{
    CHECK(is_about_cxx());
    CHECK(square(2) == 4);
    CHECK(area(2, 4) == 8);
}
```

function-types.cpp

🛠️ Functions

Functions different in syntax. There are two types of function syntax we will use in this course. You can use either, just make sure you're consistent.

```cpp
#include <iostream>

auto main() -> int
{
    // put "Hello world\n" to the character output
    std::cout << "Hello, world!\n";
}

/*#include <iostream>

int main() {
  // put "Hello world\n" to the character output
  std::cout << "Hello, world!\n";
}*/
```

function-syntax.cpp

🛠️ Functions

Default arguments

- Functions can use default arguments, which is used if an actual argument is not specified when a function is called
- Default values are used for the trailing parameters of a function call - this means that ordering is important
- Formal parameters: Those that appear in function definition
- Actual parameters (arguments): Those that appear when calling the function

🛠️ Functions

Default arguments

```cpp
#include <string>

std::string rgb(short r = 0, short g = 0, short b = 0)
{
    (void)r;
    (void)g;
    (void)b;
    return "";
}

int main()
{
    rgb(); // rgb(0, 0, 0);
    rgb(100); // Rgb(100, 0, 0);
    rgb(100, 200); // Rgb(100, 200, 0)
    // rgb(100, , 200);   // error
}
```

function-parameters.cpp

🛠️ Functions

Function overloading

- Function overloading refers to a family of functions in the same scope that have the same name but different formal parameters.
- This can make code easier to write and understand

```cpp
#include <catch2/catch.hpp>

auto square(int const x) -> int
{
    return x * x;
}

auto square(double const x) -> double
{
    return x * x;
}

TEST_CASE()
{

    CHECK(square(2) == 4);
    CHECK(square(2.0) == 4.0);
    CHECK(square(2.0) != 4);
}
```

function-overloading.cpp

🛠️ Functions

Function overloading **resolution**

- This is the process of "function matching"
  - Step 1: Find candidate functions: Same name
  - Step 2: Select viable ones: Same number arguments + each argument convertible
  - Step 3: Find a best-match: Type much better in at least one argument

🛠️ Functions

Function overloading **resolution**

Errors in function matching are found during compile time. Return types are ignored.

Read more about this here.

```cpp
/*
auto g() -> void;
auto f(int) -> void;
auto f(int, int) -> void;
auto f(double, double = 3.14) -> void;
f(5.6); // calls f(double, double)
*/
```

function-overloading-resolution.cpp

- When writing code, try and only create overloads that are trivial
  - If non-trivial to understand, name your functions differently

🛣️ If Statement

If statements are what you'd expect

```cpp
#include <catch2/catch.hpp>

auto collatz_point_if_statement(int const x) -> int
{
    if (x % 2 == 0) {
        return x / 2;
    }
    return 3 * x + 1;
}

TEST_CASE()
{
    CHECK(collatz_point_if_statement(6) == 3);
    CHECK(collatz_point_if_statement(5) == 16);
}
```

if-basic.cpp

🛣️ If Statement

We also have short-hand conditional expressions

```cpp
#include <catch2/catch.hpp>

auto is_even(int const x) -> bool
{
    return x % 2 == 0;
}

auto collatz_point_conditional(int const x) -> int
{
    return is_even(x) ? x / 2
                      : 3 * x + 1;
}

TEST_CASE()
{
    CHECK(collatz_point_conditional(6) == 3);
    CHECK(collatz_point_conditional(5) == 16);
}
```

if-short.cpp

🛣️ If Statement

We can also do things via the switch method

```
#include <catch2/catch.hpp>

auto is_digit(char const c) -> bool
{
    switch (c) {
    case '0':
        [[fallthrough]];
    case '1':
        [[fallthrough]];
    case '2':
        [[fallthrough]];
    case '3':
        [[fallthrough]];
    case '4':
        [[fallthrough]];
    case '5':
        [[fallthrough]];
    case '6':
        [[fallthrough]];
    case '7':
        [[fallthrough]];
    case '8':
        [[fallthrough]];
    case '9':
        return true;
    default:
        return false;
    }
}

TEST_CASE()
{
    CHECK(is_digit('6'));
    CHECK(not is_digit('A'));
}
```

if-switch.cpp

🚢 Sequenced Collections

There are a number of sequenced containers we will talk about in week 2. Today we will discuss vector, a very basic sequenced container.

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{
    auto const single_digits = std::vector<int> {
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    };

    auto more_single_digits = single_digits;
    REQUIRE(single_digits == more_single_digits);

    more_single_digits[2] = 0;
    CHECK(single_digits != more_single_digits);

    more_single_digits.push_back(0);
    CHECK(more_single_digits.size() == 11);
}
```

sequenced-collections.cpp

🔬 Values & References

- We can use pointers in C++ just like C, but generally we don't want to
- A reference is an alias for another object: You can use it as you would the original object
- Similar to a pointer, but:
    - Don't need to use -> to access elements
    - Can't be null
    - You can't change what they refer to once set

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{
    auto i = 1;
    auto& j = i;
    j = 3;

    CHECK(i == 3);
}
```

references.cpp

🔬 Values & References

- A reference to const means you can't modify the object using the reference
- The object is still able to be modified, just not through this reference

```cpp
#include <iostream>

int main()
{
    auto i = 1;
    auto const& ref = i;
    std::cout << ref << '\n';
    i++; // This is fine
    std::cout << ref << '\n';
    // ref++; // This is not

    auto const j = 1;
    auto const& jref = j; // this is allowed
    // auto& ref = j; // not allowed
    std::cout << jref << "\n";
}
```

references-more.cpp

Function Passing Methods

Pass by value

The actual argument is copied into the memory being used to hold the formal parameters value during the function call/execution

```cpp
#include <iostream>

auto swap(int x, int y) -> void
{
    auto const tmp = x;
    x = y;
    y = tmp;
}

auto main() -> int
{
    auto i = 1;
    auto j = 2;
    std::cout << i << ' ' << j << '\n'; // prints 1 2
    swap(i, j);
    std::cout << i << ' ' << j << '\n'; // prints 1 2... not swapped?
}
```

pass-by-value.cpp

🖇 Function Passing Methods

Pass by reference

- The formal parameter merely acts as an alias for the actual parameter
- Anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter
- Pass by reference is useful when:
  - The argument has no copy operation
  - The argument is large

Function Passing Methods

Pass by reference

```cpp
#include <iostream>

auto swap(int& x, int& y) -> void
{
    auto const tmp = x;
    x = y;
    y = tmp;
}

auto main() -> int
{
    auto i = 1;
    auto j = 2;
    std::cout << i << ' ' << j << '\n'; // 1 2
    swap(i, j);
    std::cout << i << ' ' << j << '\n'; // 2 1
}
```

pass-by-reference-new.cpp

```c
// C equivalent
#include <stdio.h>

void swap(int* x, int* y)
{
    auto const tmp = *x;
    *x = *y;
    *y = tmp;
}

int main()
{
    int i = 1;
    int j = 2;
    printf("%d %d\n", i, j);
    swap(&i, &j);
    printf("%d %d\n", i, j);
}
```

pass-by-reference-old.cpp

Function Passing Methods

Comparing value & reference performance

```cpp
/*auto by_value(std::string const sentence) -> char;
// takes ~153.67 ns
by_value(two_kb_string);

auto by_reference(std::string const& sentence) -> char;
// takes ~8.33 ns
by_reference(two_kb_string);

auto by_value(std::vector<std::string> const long_strings) -> char;
// takes ~2'920 ns
by_value(sixteen_two_kb_strings);

auto by_reference(std::vector<std::string> const& long_strings) -> char;
// takes ~13 ns
by_reference(sixteen_two_kb_strings);*/
```

value-reference-performance.cpp

🤔 Declarations V Definitions

- A declaration makes known the type and the name of a variable
- A definition is a declaration, but also does extra things
  - A variable definition allocates storage for, and constructs a variable
  - A class definition allows you to create variables of the class' type
  - You can call functions with only a declaration, but must provide a definition later
- Everything must have precisely one definition

```cpp
#include <catch2/catch.hpp>

void declared_fn(int arg);
class declared_type;

// This class is defined, but not all the methods are.
class defined_type {
    int declared_member_fn(double);
    int defined_member_fn(int arg) { return arg; }
};

// These are all defined.
int defined_fn(int arg)
{
    (void)arg;
    return 1;
}

TEST_CASE()
{
    int i;
    int const j = 1;
    auto vd = std::vector<double> {};
    (void)i;
    (void)j;
    (void)vd;
}
```

declaration-definition.cpp

➿ Looping

For-range Statements

```cpp
#include <string>
#include <vector>

auto all_computer_scientists(std::vector<std::string> const& names) -> bool
{
    auto const famous_mathematician = std::string("Gauss");
    auto const famous_physicist = std::string("Newton");

    for (auto const& name : names) {
        if (name == famous_mathematician or name == famous_physicist) {
            return false;
        }
    }

    return true;
}

int main()
{
}
```

for-range.cpp

➿ Looping

For Statements

```cpp
auto square(int n)
{
    return n * n;
}

auto cube(int n)
{
    return n * n * n;
}

auto square_vs_cube() -> bool
{
    // 0 and 1 are special cases, since they're actually equal.
    if (square(0) != cube(0) or square(1) != cube(1)) {
        return false;
    }

    for (auto i = 2; i < 100; ++i) {
        if (square(i) == cube(i)) {
            return false;
        }
    }

    return true;
}
```

for-statements.cpp

🧑 Enumarations

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{
    enum class computing_courses {
        intro,
        data_structures,
        engineering_design,
        compilers,
        cplusplus,
    };

    auto const computing101 = computing_courses::intro;
    auto const computing102 = computing_courses::data_structures;
    CHECK(computing101 != computing102);
}
```

enumerations.cpp

📦 Hash Maps

```cpp
#include <catch2/catch.hpp>
#include <string>
#include <unordered_map>

auto check_code_mapping(
    std::unordered_map<std::string, std::string> const& country_codes,
    std::string const& code,
    std::string const& name) -> void
{
    auto const country = country_codes.find(code);
    REQUIRE(country != country_codes.end());

    auto const [key, value] = *country;
    CHECK(code == key);
    CHECK(name == value);
}

TEST_CASE()
{
    auto country_codes = std::unordered_map<std::string, std::string> {
        { "AU", "Australia" },
        { "NZ", "New Zealand" },
        { "CK", "Cook Islands" },
        { "ID", "Indonesia" },
        { "DK", "Denmark" },
        { "CN", "China" },
        { "JP", "Japan" },
        { "ZM", "Zambia" },
        { "YE", "Yemen" },
        { "CA", "Canada" },
        { "BR", "Brazil" },
        { "AQ", "Antarctica" },
    };
    CHECK(country_codes.contains("AU"));
    CHECK(not country_codes.contains("DE")); // Germany not present
    country_codes.emplace("DE", "Germany");
    CHECK(country_codes.contains("DE"));
}
```

hash-map.cpp

😵 Program Errors

There are 4 types of program errors:

- Compile-time
- Link-time
- Run-time
- Logic

😵 Program Errors

Compile-time

```cpp
auto main() -> int
{
    // a = 5; // Compile-time error: type not specified
    // (void) a;
}
```

error-compile.cpp

😵 Program Errors

Link-time

```cpp
#include <iostream>

auto is_cs6771() -> bool;

int main()
{
    // std::cout << is_cs6771() << "\n";
}
```

error-link.cpp

😵 Program Errors

Run-time

```cpp
#include <fstream>
#include <stdexcept>

int main()
{
    // attempting to open a file...
    if (auto file = std::ifstream("hello.txt"); not file) {
        throw std::runtime_error("Error: file not found.\n");
    }
}
```

error-runtime.cpp

😵 Program Errors

Logic

```cpp
#include <catch2/catch.hpp>

TEST_CASE()
{
    auto const empty = std::string("");
    CHECK(empty[0] == 'C'); // Logic error: bad character access
}
```

error-logic.cpp

📁 File Input & Output

```cpp
#include <fstream>
#include <iostream>

int main()
{
    // Below line only works C++17
    std::ofstream fout { "data.out" };
    if (auto in = std::ifstream { "data.in" }; in) { // attempts to open file, checks it was opened
        for (auto i = 0; in >> i;) { // reads in
            std::cout << i << '\n';
            fout << i;
        }
        if (in.bad()) {
            std::cerr << "unrecoverable error (e.g. disk disconnected?)\n";
        } else if (not in.eof()) {
            std::cerr << "bad input: didn't read an int\n";
        }
    } // closes file automatically <-- no need to close manually!
    else {
        std::cerr << "unable to read data.in\n";
    }
    fout.close();
}
```

file-io.cpp

🧪 Gitlab Merge Requests

Random note! Don't forget to check gitlab for merge requests we push out.

👂 Feedback

Or go to the form here.