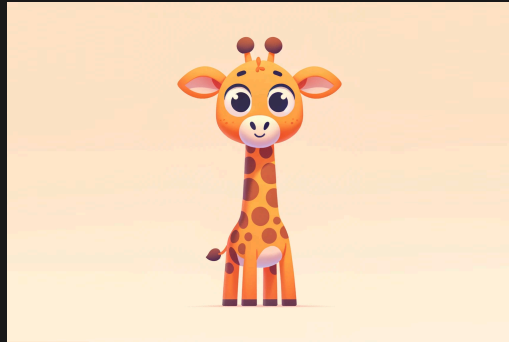


COMP6771

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

```
1 int main()
2 {
3     // `int` for integers.
4     int meaning_of_life = 42;
5
6     // `double` for rational numbers.
7     double six_feet_in_metres = 1.8288;
8
9     // `char` for single characters.
10    char letter = 'C';
11
12    (void)meaning_of_life;
13    (void)six_feet_in_metres;
14    (void)letter;
15 }
```

basic1.cpp

Basic Types!

As well as many more types

```
1 #include <string>
2
3 int main()
4 {
5     // `string` for text.
6     std::string course_code = std::string("COMP6771");
7
8     // `bool` for truth
9     bool is_cxx = true;
10    bool is_danish = false;
11
12    (void)is_cxx;
13    (void)is_danish;
14 }
```

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.

```
1 #include <iostream>
2 #include <limits>
3
4 int main()
5 {
6     std::cout << std::numeric_limits<int>::max() << "\n";
7     std::cout << std::numeric_limits<int>::min() << "\n";
8     std::cout << std::numeric_limits<double>::max() << "\n";
9     std::cout << std::numeric_limits<double>::min() << "\n";
10 }
```

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.

```
1 int main()  
2 {  
3     auto i = 0; // i is an int  
4     auto j = 8.5; // j is a double  
5     auto k = false; // k is a bool  
6     (void)i;  
7     (void)j;  
8     (void)k;  
9 }
```

`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

```
1 int main()
2 {
3     // `int` for integers.
4     auto const meaning_of_life = 42;
5
6     // `double` for rational numbers.
7     auto const six_feet_in_metres = 1.8288;
8
9     (void)meaning_of_life;
10    (void)six_feet_in_metres;
11
12    // meaning_of_life++; // COMPILER ERROR HERE
13 }
```

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

```
1 int main()  
2 {  
3     auto const meaning_of_life1 = 42; // good  
4     const auto meaning_of_life2 = 42; // bad  
5     (void)meaning_of_life1;  
6     (void)meaning_of_life2;  
7 }
```

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

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5     auto const x = 10;
6     auto const y = 173;
7
8     auto const sum = 183;
9     CHECK(x + y == sum);
10
11     auto const difference = 163;
12     CHECK(y - x == difference);
13     CHECK(x - y == -difference);
14
15     auto const product = 1730;
16     CHECK(x * y == product);
17
18     auto const quotient = 17;
19     CHECK(y / x == quotient);
20
21     auto const remainder = 3;
22     CHECK(y % x == remainder);
23 }
```

expression-integral.cpp



Expressions

Floating Points

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5     auto const x = 15.63;
6     auto const y = 1.23;
7
8     auto const sum = 16.86;
9     CHECK(x + y == sum);
10
11     auto const difference = 14.4;
12     CHECK(x - y == difference);
13     CHECK(y - x == -difference);
14
15     auto const product = 19.2249;
16     CHECK(x * y == product);
17
18     auto const expected = 12.7073170732;
19     auto const actual = x / y;
20     auto const acceptable_delta = 0.0000001;
21     CHECK(std::abs(expected - actual) < acceptable_delta);
22 }
```

expression-floating.cpp



Expressions

Strings

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5
6     auto const expr = std::string("Hello, expressions!");
7     auto const cxx = std::string("Hello, C++!");
8
9     CHECK(expr != cxx);
10    CHECK(expr.front() == cxx[0]);
11
12    auto expr2 = expr;
13
14    // Abort TEST_CASE if expression is false
15    REQUIRE(expr == expr2);
16 }
```

expression-string.cpp



Expressions

Booleans

```
1 #include <catch2/catch.hpp>
2
3 auto const is_comp6771 = true;
4 auto const is_about_cxx = true;
5 auto const is_about_german = false;
6
7 TEST_CASE()
8 {
9     CHECK((is_comp6771 and is_about_cxx));
10    CHECK((is_about_german or is_about_cxx));
11    CHECK(not is_about_german);
12 }
13
14 // You can use classic && or || as well
```

[expression-boolean.cpp](#)



C++ Has Value Semantics

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5     auto const hello = std::string("Hello!");
6     auto hello2 = hello;
7
8     // Abort TEST_CASE if expression is false
9     REQUIRE(hello == hello2);
10
11     hello2.append("2");
12     REQUIRE(hello != hello2);
13
14     CHECK(hello.back() == '!');
15     CHECK(hello2.back() == '2');
16 }
```

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

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5     auto const i = 0;
6     {
7         auto d = 0.0;
8         REQUIRE(d == 0.0);
9
10        d = i; // Silent conversion from int to double
11        CHECK(d == 42.0);
12        CHECK(d != 41);
13    }
14 }
```

[convert-implicit.cpp](#)



Type Conversion

Explicit promoting conversions

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5     auto const i = 0;
6     {
7         // Preferred over implicit, since your intention is clear
8         auto const d = static_cast<double>(i);
9         CHECK(d == 42.0);
10        CHECK(d != 41);
11    }
12 }
```

[convert-explicit.cpp](#)



Functions

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



Functions

Different types of functions

```
1 #include <catch2/catch.hpp>
2
3 bool is_about_cxx()
4 { // nullary functions (no parameters)
5     return true;
6 }
7
8 int square(int const x)
9 { // unary functions (one parameter)
10     return x * x;
11 }
12
13 int area(int const width, int const length)
14 { // binary functions (two parameters)
15     return width * length;
16 }
17
18 TEST_CASE()
19 {
20     CHECK(is_about_cxx());
21     CHECK(square(2) == 4);
22     CHECK(area(2, 4) == 8);
23 }
```

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.

```
1 #include <iostream>
2
3 auto main() -> int
4 {
5     // put "Hello world\n" to the character output
6     std::cout << "Hello, world!\n";
7 }
8
9 /*#include <iostream>
10
11 int main() {
12     // put "Hello world\n" to the character output
13     std::cout << "Hello, world!\n";
14 }*/
```

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

```
1 #include <string>
2
3 std::string rgb(short r = 0, short g = 0, short b = 0)
4 {
5     (void)r;
6     (void)g;
7     (void)b;
8     return "";
9 }
10
11 int main()
12 {
13     rgb(); // rgb(0, 0, 0);
14     rgb(100); // Rgb(100, 0, 0);
15     rgb(100, 200); // Rgb(100, 200, 0)
16     // rgb(100, , 200); // error
17 }
```

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

```
1 #include <catch2/catch.hpp>
2
3 auto square(int const x) -> int
4 {
5     return x * x;
6 }
7
8 auto square(double const x) -> double
9 {
10    return x * x;
11 }
12
13 TEST_CASE()
14 {
15
16     CHECK(square(2) == 4);
17     CHECK(square(2.0) == 4.0);
18     CHECK(square(2.0) != 4);
19 }
```

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](#).

```
1 /*
2 auto g() -> void;
3 auto f(int) -> void;
4 auto f(int, int) -> void;
5 auto f(double, double = 3.14) -> void;
6 f(5.6); // calls f(double, double)
7 */
```

[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

```
1 #include <catch2/catch.hpp>
2
3 auto collatz_point_if_statement(int const x) -> int
4 {
5     if (x % 2 == 0) {
6         return x / 2;
7     }
8     return 3 * x + 1;
9 }
10
11 TEST_CASE()
12 {
13     CHECK(collatz_point_if_statement(6) == 3);
14     CHECK(collatz_point_if_statement(5) == 16);
15 }
```

if-basic.cpp



If Statement

We also have short-hand conditional expressions

```
1 #include <catch2/catch.hpp>
2
3 auto is_even(int const x) -> bool
4 {
5     return x % 2 == 0;
6 }
7
8 auto collatz_point_conditional(int const x) -> int
9 {
10     return is_even(x) ? x / 2
11                     : 3 * x + 1;
12 }
13
14 TEST_CASE()
15 {
16     CHECK(collatz_point_conditional(6) == 3);
17     CHECK(collatz_point_conditional(5) == 16);
18 }
```

if-short.cpp



If Statement

We can also do things via the switch method

```
1 #include <catch2/catch.hpp>
2
3 auto is_digit(char const c) -> bool
4 {
5     switch (c) {
6     case '0':
7         [[fallthrough]];
8     case '1':
9         [[fallthrough]];
10    case '2':
11        [[fallthrough]];
12    case '3':
13        [[fallthrough]];
14    case '4':
15        [[fallthrough]];
16    case '5':
17        [[fallthrough]];
18    case '6':
19        [[fallthrough]];
20    case '7':
21        [[fallthrough]];
22    case '8':
23        [[fallthrough]];
24    case '9':
25        return true;
26    default:
27        return false;
28    }
29 }
30
31 TEST_CASE()
32 {
33     CHECK(is_digit('6'));
34     CHECK(not is_digit('A'));
35 }
```

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.

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5     auto const single_digits = std::vector<int> {
6         0, 1, 2, 3, 4, 5, 6, 7, 8, 9
7     };
8
9     auto more_single_digits = single_digits;
10    REQUIRE(single_digits == more_single_digits);
11
12    more_single_digits[2] = 0;
13    CHECK(single_digits != more_single_digits);
14
15    more_single_digits.push_back(0);
16    CHECK(more_single_digits.size() == 11);
17 }
```

[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

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5     auto i = 1;
6     auto& j = i;
7     j = 3;
8
9     CHECK(i == 3);
10 }
```

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

```
1 #include <iostream>
2
3 int main()
4 {
5     auto i = 1;
6     auto const& ref = i;
7     std::cout << ref << '\n';
8     i++; // This is fine
9     std::cout << ref << '\n';
10    // ref++; // This is not
11
12    auto const j = 1;
13    auto const& jref = j; // this is allowed
14    // auto& ref = j; // not allowed
15    std::cout << jref << "\n";
16 }
```

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

```
1 #include <iostream>
2
3 auto swap(int x, int y) -> void
4 {
5     auto const tmp = x;
6     x = y;
7     y = tmp;
8 }
9
10 auto main() -> int
11 {
12     auto i = 1;
13     auto j = 2;
14     std::cout << i << ' ' << j << '\n'; // prints 1 2
15     swap(i, j);
16     std::cout << i << ' ' << j << '\n'; // prints 1 2... not swapped?
17 }
```

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

```
1 #include <iostream>
2
3 auto swap(int& x, int& y) -> void
4 {
5     auto const tmp = x;
6     x = y;
7     y = tmp;
8 }
9
10 auto main() -> int
11 {
12     auto i = 1;
13     auto j = 2;
14     std::cout << i << ' ' << j << '\n'; // 1 2
15     swap(i, j);
16     std::cout << i << ' ' << j << '\n'; // 2 1
17 }
```

pass-by-reference-new.cpp

```
1 // C equivalent
2 #include <stdio.h>
3
4 void swap(int* x, int* y)
5 {
6     auto const tmp = *x;
7     *x = *y;
8     *y = tmp;
9 }
10
11 int main()
12 {
13     int i = 1;
14     int j = 2;
15     printf("%d %d\n", i, j);
16     swap(&i, &j);
17     printf("%d %d\n", i, j);
18 }
```

pass-by-reference-old.cpp



Function Passing Methods

Comparing value & reference performance

```
1 /*auto by_value(std::string const sentence) -> char;
2 // takes ~153.67 ns
3 by_value(two_kb_string);
4
5 auto by_reference(std::string const& sentence) -> char;
6 // takes ~8.33 ns
7 by_reference(two_kb_string);
8
9 auto by_value(std::vector<std::string> const long_strings) -> char;
10 // takes ~2'920 ns
11 by_value(sixteen_two_kb_strings);
12
13 auto by_reference(std::vector<std::string> const& long_strings) -> char;
14 // takes ~13 ns
15 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

```
1 #include <catch2/catch.hpp>
2
3 void declared_fn(int arg);
4 class declared_type;
5
6 // This class is defined, but not all the methods are.
7 class defined_type {
8     int declared_member_fn(double);
9     int defined_member_fn(int arg) { return arg; }
10 };
11
12 // These are all defined.
13 int defined_fn(int arg)
14 {
15     (void)arg;
16     return 1;
17 }
18
19 TEST_CASE()
20 {
21     int i;
22     int const j = 1;
23     auto vd = std::vector<double> {};
24     (void)i;
25     (void)j;
26     (void)vd;
27 }
```

declaration-definition.cpp

Looping

For-range Statements

```
1 #include <string>
2 #include <vector>
3
4 auto all_computer_scientists(std::vector<std::string> const& names) -> bool
5 {
6     auto const famous_mathematician = std::string("Gauss");
7     auto const famous_physicist = std::string("Newton");
8
9     for (auto const& name : names) {
10         if (name == famous_mathematician or name == famous_physicist) {
11             return false;
12         }
13     }
14
15     return true;
16 }
17
18 int main()
19 {
20 }
```

for-range.cpp

Looping

For Statements

```
1 auto square(int n)
2 {
3     return n * n;
4 }
5
6 auto cube(int n)
7 {
8     return n * n * n;
9 }
10
11 auto square_vs_cube() -> bool
12 {
13     // 0 and 1 are special cases, since they're actually equal.
14     if (square(0) != cube(0) or square(1) != cube(1)) {
15         return false;
16     }
17
18     for (auto i = 2; i < 100; ++i) {
19         if (square(i) == cube(i)) {
20             return false;
21         }
22     }
23
24     return true;
25 }
```

for-statements.cpp



Enumerations

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5     enum class computing_courses {
6         intro,
7         data_structures,
8         engineering_design,
9         compilers,
10        cplusplus,
11    };
12
13    auto const computing101 = computing_courses::intro;
14    auto const computing102 = computing_courses::data_structures;
15    CHECK(computing101 != computing102);
16 }
```

[enumerations.cpp](#)



Hash Maps

```
1 #include <catch2/catch.hpp>
2 #include <string>
3 #include <unordered_map>
4
5 auto check_code_mapping(
6     std::unordered_map<std::string, std::string> const& country_codes,
7     std::string const& code,
8     std::string const& name) -> void
9 {
10     auto const country = country_codes.find(code);
11     REQUIRE(country != country_codes.end());
12
13     auto const [key, value] = *country;
14     CHECK(code == key);
15     CHECK(name == value);
16 }
17
18 TEST_CASE()
19 {
20     auto country_codes = std::unordered_map<std::string, std::string> {
21         { "AU", "Australia" },
22         { "NZ", "New Zealand" },
23         { "CK", "Cook Islands" },
24         { "ID", "Indonesia" },
25         { "DK", "Denmark" },
26         { "CN", "China" },
27         { "JP", "Japan" },
28         { "ZM", "Zambia" },
29         { "YE", "Yemen" },
30         { "CA", "Canada" },
31         { "BR", "Brazil" },
32         { "AQ", "Antarctica" },
33     };
34     CHECK(country_codes.contains("AU"));
35     CHECK(not country_codes.contains("DE")); // Germany not present
36     country_codes.emplace("DE", "Germany");
37     CHECK(country_codes.contains("DE"));
38 }
```

hash-map.cpp

Program Errors

There are 4 types of program errors:

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

Program Errors

Compile-time

```
1 auto main() -> int
2 {
3     // a = 5; // Compile-time error: type not specified
4     // (void) a;
5 }
```

[error-compile.cpp](#)

Program Errors

Link-time

```
1 #include <iostream>
2
3 auto is_cs6771() -> bool;
4
5 int main()
6 {
7     // std::cout << is_cs6771() << "\n";
8 }
```

error-link.cpp

Program Errors

Run-time

```
1 #include <fstream>
2 #include <stdexcept>
3
4 int main()
5 {
6     // attempting to open a file...
7     if (auto file = std::ifstream("hello.txt"); not file) {
8         throw std::runtime_error("Error: file not found.\n");
9     }
10 }
```

error-runtime.cpp

Program Errors

Logic

```
1 #include <catch2/catch.hpp>
2
3 TEST_CASE()
4 {
5     auto const empty = std::string("");
6     CHECK(empty[0] == 'C'); // Logic error: bad character access
7 }
```

error-logic.cpp

File Input & Output

```
1 #include <fstream>
2 #include <iostream>
3
4 int main()
5 {
6     // Below line only works C++17
7     std::ofstream fout { "data.out" };
8     if (auto in = std::ifstream { "data.in" }; in) { // attempts to open file, checks it was opened
9         for (auto i = 0; in >> i;) { // reads in
10             std::cout << i << '\n';
11             fout << i;
12         }
13         if (in.bad()) {
14             std::cerr << "unrecoverable error (e.g. disk disconnected?)\n";
15         } else if (not in.eof()) {
16             std::cerr << "bad input: didn't read an int\n";
17         }
18     } // closes file automatically <-- no need to close manually!
19     else {
20         std::cerr << "unable to read data.in\n";
21     }
22     fout.close();
23 }
```

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](#).

