

COMP6771

Advanced C++ Programming

Week 1.3

C++ Basics

Basic types

Types have defined storage requirements and behaviours. C++ has a number of standard types you're familiar with from C, but then also many more!

Basic types

```
1 // `int` for integers.
2 int meaning_of_life = 42;
3
4 // `double` for rational numbers.
5 double six_feet_in_metres = 1.8288;
6
7 // report if this expression is false
8 CHECK(six_feet_in_metres < meaning_of_life);
9
10 // `string` for text.
11 std::string course_code = std::string("COMP6771");
12
13 // `char` for single characters.
14 char letter = 'C';
15
16 CHECK(course_code.front() == letter);
17
18 // `bool` for truth
19 bool is_cxx = true;
20 bool is_danish = false;
21
22 CHECK(is_cxx != is_danish);
```

demo100-types.cpp

Basic types

Remember that C++ runs directly on hardware, which means the value of some types may differ depending on the system.

An example of a library you can include to display these are below:

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

demo101-limits.cpp

Auto

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 #include <iostream>
2 #include <limits>
3 #include <vector>
4
5 int main() {
6     auto i = 0; // i is an int
7     auto j = 8.5; // j is a double
8     // auto k; // this would not work as there is no RHS to infer from
9     std::vector<int> f;
10    auto k = f; // k is std::vector<int>
11    k.push_back(5);
12    std::cout << k.size() << "\n";
13 }
```

demo102-auto.cpp

Const

- 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
- We try and use east-const in this course (const on the right)

Const

```
1 // `int` for integers.
2 auto const meaning_of_life = 42;
3
4 // `double` for rational numbers.
5 auto const six_feet_in_metres = 1.8288;
6
7 meaning_of_life++; NOT ALLOWED - compile error
8
9 // report if this expression is false
10 CHECK(six_feet_in_metres < meaning_of_life);
```

demo103-const.cpp

Why Const

- 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++

Integral expressions

```
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);
```

Floating-point expressions

```
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);
```

String expressions

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

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

```
auto const concat = absl::StrCat(expr, " ", cxx);  
CHECK(concat == "Hello, expressions! Hello, C++!");
```

```
auto expr2 = expr;
```

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

demo104-expressions.cpp

Boolean expressions

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

```
CHECK((is_comp6771 and is_about_cxx));
```

```
CHECK((is_about_german or is_about_cxx));
```

```
CHECK(not is_about_german);
```

demo104-expressions.cpp

- You can use classic && or || as well

C++ has value semantics

```
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');
```

demo105-value.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.

Implicit promoting conversions

```
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);
}
```

demo106-conversions.cpp

Explicit promoting conversions

```
auto const i = 0;
```

demo106-conversions.cpp

Explicit promoting conversions

```
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);
}
```

demo106-conversions.cpp

Functions

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

Function Types

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

```
int square(int const x) { // unary functions (one parameter)
    return x * x;
}
CHECK(square(2) == 4);
```

```
int area(int const width, int const length) { // binary functions (two parameters)
    return width * length;
}
CHECK(area(2, 4) == 8);
```

demo107-functions.cpp

Function 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     // put "Hello world\n" to the character output
5     std::cout << "Hello, world!\n";
6 }
```

```
1 #include <iostream>
2
3 int main() {
4     // put "Hello world\n" to the character output
5     std::cout << "Hello, world!\n";
6 }
```

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

```
std::string rgb(short r = 0, short g = 0, short b = 0);  
rgb(); // rgb(0, 0, 0);  
rgb(100); // Rgb(100, 0, 0);  
rgb(100, 200); // Rgb(100, 200, 0)  
rgb(100, , 200); // error
```

demo107-functions.cpp

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

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

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

Overload 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

Errors in function matching are found during compile time

Return types are ignored. Read more about this [here](#).

```
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)
```

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

if-statement

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

```
CHECK(collatz_point_if_statement(6) == 3);  
CHECK(collatz_point_if_statement(5) == 16);
```

demo108-selection.cpp

short-hand conditional expressions

```
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;  
}
```

```
CHECK(collatz_point_conditional(6) == 3);  
CHECK(collatz_point_conditional(5) == 16);
```

demo108-selection.cpp

switch-statement

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

demo108-selection.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.

```
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);
```

demo109-vector.cpp

Sequenced collections

```
auto const single_digits = std::vector<int>{
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9
};
more_single_digits.push_back(0);
CHECK(ranges::count(more_single_digits, 0) == 2);
more_single_digits.pop_back();
CHECK(ranges::count(more_single_digits, 0) == 1);
CHECK(std::erase(more_single_digits, 0) == 1);
CHECK(ranges::count(more_single_digits, 0) == 0);
CHECK(ranges::distance(more_single_digits) == 8);
```

demo109-vector.cpp

Values and 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

```
auto i = 1;  
auto& j = i;  
j = 3;
```

```
CHECK(i == 3)
```

demo110-references.cpp

References and const

- 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

```
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
```

demo110-references.cpp

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

demo111-pass1.cpp

Functions: 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

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

demo111-pass2.cpp

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

Values and references

```
auto by_value(std::string const sentence) -> char;
```

Values and references

```
auto by_value(std::string const sentence) -> char;
```

```
auto by_reference(std::string const& sentence) -> char;
```

```
auto by_value(std::vector<std::string> const long_strings) -> char;
```

```
auto by_reference(std::vector<std::string> const& long_strings) -> char;
```

Values and references

```
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;
```

```
auto by_value(std::vector<std::string> const long_strings) -> char;
```

```
auto by_reference(std::vector<std::string> const& long_strings) -> char;
```

Values and references

```
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;
```

```
auto by_reference(std::vector<std::string> const& long_strings) -> char;
```

Values and references

```
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;
```

Values and references

```
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);
```

Declarations vs 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 void declared_fn(int arg);
2 class declared_type;
3
4 // This class is defined, but not all the methods are.
5 class defined_type {
6     int declared_member_fn(double);
7     int defined_member_fn(int arg) { return arg; }
8 };
9
10 // These are all defined.
11 int defined_fn() { return 1; }
12
13 int i;
14 int const j = 1;
15 auto vd = std::vector<double>{};
```


range-for-statements

```
1 auto all_computer_scientists(std::vector<std::string> const& names) -> bool {
2     auto const famous_mathematician = std::string("Gauss");
3     auto const famous_physicist = std::string("Newton");
4
5     for (auto const& name : names) {
6         if (name == famous_mathematician or name == famous_physicist) {
7             return false;
8         }
9     }
10
11     return true;
12 }
```

demo112-iteration.cpp

for-statements

```
1 auto square_vs_cube() -> bool {
2     // 0 and 1 are special cases, since they're actually equal.
3     if (square(0) != cube(0) or square(1) != cube(1)) {
4         return false;
5     }
6
7     for (auto i = 2; i < 100; ++i) {
8         if (square(i) == cube(i)) {
9             return false;
10        }
11    }
12
13    return true;
14 }
```

demo112-iteration.cpp

User-defined types: enumerations

```
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);
```

demo113-enum.cpp

Hash sets

```
auto computer_scientists = std::unordered_set<std::string>{
    "Lovelace",
    "Babbage",
    "Turing",
    "Hamilton",
    "Church",
    "Borg",
};

CHECK(computer_scientists.contains("Lovelace"));
CHECK(not computer_scientists.contains("Gauss"));

1 computer_scientists.insert("Gauss");
2 CHECK(computer_scientists.contains("Gauss"));
3
4 computer_scientists.erase("Gauss");
5 CHECK(not computer_scientists.contains("Gauss"));
```

Hash sets

```
auto computer_scientists = std::unordered_set<std::string>{
    "Lovelace",
    "Babbage",
    "Turing",
    "Hamilton",
    "Church",
    "Borg",
};

CHECK(computer_scientists.contains("Lovelace"));
CHECK(not computer_scientists.contains("Gauss"));

1 computer_scientists.insert("Gauss");
2 CHECK(computer_scientists.contains("Gauss"));
3
4 computer_scientists.erase("Gauss");
5 CHECK(not computer_scientists.contains("Gauss"));
```

Finding an element & Empty Set

```
auto ada = computer_scientists.find("Lovelace");  
REQUIRE(ada != computer_scientists.end());  
  
CHECK(*ada == "Lovelace");
```

```
1 computer_scientists.clear();  
2 CHECK(computer_scientists.empty());
```

demo114-set.cpp

Finding an element & Empty Set

```
auto ada = computer_scientists.find("Lovelace");  
REQUIRE(ada != computer_scientists.end());  
  
CHECK(*ada == "Lovelace");
```

```
1 computer_scientists.clear();  
2 CHECK(computer_scientists.empty());
```

demo114-set.cpp

Hash maps

```
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"));
```

demo115-map.cpp

Hash maps

```
1 auto check_code_mapping(  
2     std::unordered_map<std::string, std::string> const& country_codes,  
3     std::string const& code,  
4     std::string const& name) -> void {  
5         auto const country = country_codes.find(code);  
6         REQUIRE(country != country_codes.end());  
7  
8         auto const [key, value] = *country;  
9         CHECK(code == key);  
10        CHECK(name == value);  
11 }
```

demo115-map.cpp

Hash maps

```
1 auto check_code_mapping(  
2     std::unordered_map<std::string, std::string> const& country_codes,  
3     std::string const& code,  
4     std::string const& name) -> void {  
5         auto const country = country_codes.find(code);  
6         REQUIRE(country != country_codes.end());  
7  
8         auto const [key, value] = *country;  
9         CHECK(code == key);  
10        CHECK(name == value);  
11 }
```

demo115-map.cpp

Hash maps

```
1 auto check_code_mapping(  
2     std::unordered_map<std::string, std::string> const& country_codes,  
3     std::string const& code,  
4     std::string const& name) -> void {  
5         auto const country = country_codes.find(code);  
6         REQUIRE(country != country_codes.end());  
7  
8         auto const [key, value] = *country;  
9         CHECK(code == key);  
10        CHECK(name == value);  
11 }
```

demo115-map.cpp

Hash maps

```
1 auto check_code_mapping(  
2     std::unordered_map<std::string, std::string> const& country_codes,  
3     std::string const& code,  
4     std::string const& name) -> void {  
5         auto const country = country_codes.find(code);  
6         REQUIRE(country != country_codes.end());  
7  
8         auto const [key, value] = *country;  
9         CHECK(code == key);  
10        CHECK(name == value);  
11 }
```

demo115-map.cpp

Type Templates

Type	What it stores	Common usages
<code>std::optional<T></code>	0 or 1 T's	A function that may fail
<code>std::vector<T></code>	Any number of T's	Standard "list" type
<code>std::unordered_map<KeyT, ValueT></code>	Many Key / Value pairs	Standard "hash table" / "map" / "dictionary" type

- Later on, we will introduce a few other types
- There are other types you could use instead of `std::vector` and `std::unordered_map`, but these are good defaults
 - There are container types for linked lists, for example (but linked lists are terrible and should rarely be used)
- These are **NOT** the same as Java's generics, even though they are similar syntax to use
 - `std::vector<int>` and `std::vector<string>` are 2 different types (unlike Java, if you're familiar with it)
- We will discuss how this works when we discuss templates in later weeks

Program errors

There are 4 types of program errors that we will discuss

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

Compile-time Errors

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

Link-time Errors

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


Run-time errors

```
1 // attempting to open a file...
2 if (auto file = std::ifstream("hello.txt"); not file) {
3     throw std::runtime_error("Error: file not found.\n");
4 }
```

Logic (programmer) Errors

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

Logic (programmer) Errors

```
1 auto const s = std::string("");  
2 assert(not s.empty());  
3 CHECK(s[0] == 'C'); // Logic error: bad character access
```

File input and output

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