COMP6771
Advanced C++ Programming

Week 1.2
Intro & Types

First program

```
#include <iostream>

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

First program

```
#include <iostream>

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

Basic types

```cpp
// `int` for integers.
auto meaning_of_life = 42;

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

// report if this expression is false
CHECK(six_feet_in_metres < meaning_of_life);
```

Basic types

```
// `string` for text.
auto course_code = std::string("COMP6771");

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

CHECK(course_code.front() == letter);
```

Basic types

```
// `bool` for truth
auto is_cxx = true;
auto is_danish = false;
CHECK(is_cxx != is_danish);
```

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

Const

```
// `int` for integers.
auto const meaning_of_life = 42;

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

// report if this expression is false
CHECK(six_feet_in_metres < meaning_of_life);
```

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

Integral expressions

```
auto const x = 10;
auto const y = 173;
```

Integral expressions

```
auto const x = 10;
auto const y = 173;
auto const sum = 183;
CHECK(x + y == sum);
```

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);
```

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);
```

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);
```

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;
```

Floating-point expressions

```
auto const x = 15.63;
auto const y = 1.23;
auto const sum = 16.86;
CHECK(x + y == sum);
```

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);
```

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);
```

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

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

String expressions

```
auto const expr = std::string("Hello, expressions!");
auto const cxx = std::string("Hello, C++!");
CHECK(expr != cxx);
CHECK(expr.front() == cxx[0]);
```

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++!");
```

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);
```

C++ has value semantics

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

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

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);
```

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');
```

Boolean expressions

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

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));
```

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));
```

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);
```

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;
```

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);
}
```

Explicit promoting conversions

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

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);
}
```

Explicit narrowing (lossy) conversions

```
auto const i = 42;
```

Explicit narrowing (lossy) conversions

```
auto const i = 42;
{
  // information lost, but we're saying we know
  auto const b = gsl_lite::narrow_cast<bool>(i);
  CHECK(b == true);
  CHECK(b == gsl_lite::narrow_cast<bool>(42)); // okay
}
```

Functions

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

Nullary function (no parameters)

```cpp
auto is_about_cxx() -> bool {
  return true;
}
```

Nullary function (no parameters)

```
auto is_about_cxx() -> bool {
  return true;
}
CHECK(is_about_cxx());
```

Unary function (one parameter)

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

Unary function (one parameter)

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

```
CHECK(square(2) == 4);
```

Binary function (two parameters)

```
auto area(int const width, int const length) -> int {
  return width * length;
}
```

Binary function (two parameters)

```
auto area(int const width, int const length) -> int {
  return width * length;
}
CHECK(area(2, 4) == 8);
```

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
```

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
```

Function overloading

- Function overloading refers to a family of functions in the **same scope** that have the **same name** but **different formal parameters**.

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

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

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;
}
```

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

```
void g();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6); // calls f(double, double)
```

Overload Resolution

- This is the process of "function matching"

```
void g();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6); // calls f(double, double)
```

Overload Resolution

- This is the process of "function matching"
- Step 1: Find candidate functions: Same name

```
void g();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6); // calls f(double, double)
```

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

```
void g();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6); // calls f(double, double)
```

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

```
void g();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6); // calls f(double, double)
```

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.

```
void g();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6); // calls f(double, double)
```

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.

```
void g();
void f(int);
void f(int, int);
void f(double, double = 3.14);
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

Function overloading and const

When doing **call by value**, top-level const has no effect on the objects passed to the function. A parameter that has a top-level const is indistinguishable from the one without

Function overloading and const

When doing **call by value**, top-level const has no effect on the objects passed to the function. A parameter that has a top-level const is indistinguishable from the one without

```
// Top-level const ignored
record lookup(phone p);
record lookup(phone const p); // redeclaration

phone p;
phone const q;
lookup(p); // (1)
lookup(q); // (1)
```

```
// Low-level const not ignored
record lookup(phone& p);        // (1)
record lookup(phone const& p); // (2)

phone p;
phone const q;
lookup(p); // (1)
lookup(q); // (2)
```

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;
}
```

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);
```

if-statement

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

  return 3 * x + 1;
}
```

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);
```

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;
        }
}
```

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'));
```

Sequenced collections

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

Sequenced collections

```
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);
```

Sequenced collections

```
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);
```

Sequenced collections

```
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(ranges::count(more_single_digits, 0) == 3);
```

Sequenced collections

```
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(ranges::count(more_single_digits, 0) == 3);
more_single_digits.pop_back();
CHECK(ranges::count(more_single_digits, 0) == 2);
```

Sequenced collections

```
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(ranges::count(more_single_digits, 0) == 3);
more_single_digits.pop_back();
CHECK(ranges::count(more_single_digits, 0) == 2);
CHECK(std::erase(more_single_digits, 0) == 2);
CHECK(ranges::count(more_single_digits, 0) == 0);
CHECK(ranges::distance(more_single_digits) == 8);
```

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

Values and references

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

Values and references

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

```
// takes ~153.67 ns
by_value(two_kb_string);
```

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;
```

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);
```

Values and references

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

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

```cpp
auto by_value(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);
```

Values and references

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

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

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

```cpp
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);
```

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

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
```

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

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

```cpp
#include <iostream>

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

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

```cpp
#include <iostream>

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

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

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

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

```
#include <iostream>

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

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

range-for-statements

```
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;
}
```

for-statements

```
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;
}
```

User-defined types: enumerations

```
enum class computing_courses {
        intro,
        data_structures,
        engineering_design,
        compilers,
        cplusplus,
};
```

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);
```

User-defined types: structures

```
struct scientist {
        std::string family_name;
        std::string given_name;
        field_of_study primary_field;
        std::vector<field_of_study> secondary_fields;
};
```

Defining two objects

```
auto const famous_physicist = scientist{
    .family_name = "Newton",
    .given_name = "Isaac",
    .primary_field = field_of_study::physics,
    .secondary_fields = {field_of_study::mathematics,
                         field_of_study::astronomy,
                         field_of_study::theology},
};
```

Defining two objects

```
auto const famous_physicist = scientist{
    .family_name = "Newton",
    .given_name = "Isaac",
    .primary_field = field_of_study::physics,
    .secondary_fields = {field_of_study::mathematics,
                         field_of_study::astronomy,
                         field_of_study::theology},
};
```

```
auto const famous_mathematician = scientist{
    .family_name = "Gauss",
    .given_name = "Carl Friedrich",
    .primary_field = field_of_study::mathematics,
    .secondary_fields = {field_of_study::physics},
};
```

Member access

```
CHECK(famous_physicist.family_name
    != famous_mathematician.family_name);
```

Member access

```
CHECK(famous_physicist.family_name
   != famous_mathematician.family_name);
```

```
CHECK(famous_physicist.given_name
   != famous_mathematician.given_name);
```

Member access

```
CHECK(famous_physicist.family_name
    != famous_mathematician.family_name);
```

```
CHECK(famous_physicist.given_name
    != famous_mathematician.given_name);
```

```
CHECK(famous_physicist.primary_field
    != famous_mathematician.primary_field);
```

Member access

```
CHECK(famous_physicist.family_name
    != famous_mathematician.family_name);

CHECK(famous_physicist.given_name
    != famous_mathematician.given_name);

CHECK(famous_physicist.primary_field
    != famous_mathematician.primary_field);

CHECK(famous_physicist.secondary_fields
    != famous_mathematician.secondary_fields);
```

Wouldn't it be nicer if we could say this?

```
CHECK(famous_physicist != famous_mathematician);
```

User-defined types: structures

```
struct scientist {
        std::string family_name;
        std::string given_name;
        field_of_study primary_field;
        std::vector<field_of_study> secondary_fields;


};
```

User-defined types: structures

```
struct scientist {
        std::string family_name;
        std::string given_name;
        field_of_study primary_field;
        std::vector<field_of_study> secondary_fields;

        auto operator==(scientist const&) const -> bool = default;
};
```

Hash sets

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

Hash sets

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

```
REQUIRE(ranges::distance(computer_scientists) == 6);
CHECK(computer_scientists.contains("Lovelace"));
CHECK(not computer_scientists.contains("Gauss"));
```

Inserting an element

```
computer_scientists.insert("Gauss");
CHECK(ranges::distance(computer_scientists) == 7);
CHECK(computer_scientists.contains("Gauss"));

computer_scientists.erase("Gauss");
CHECK(ranges::distance(computer_scientists) == 6);
CHECK(not computer_scientists.contains("Gauss"));
```

Inserting an element

```
computer_scientists.insert("Gauss");
CHECK(ranges::distance(computer_scientists) == 7);
CHECK(computer_scientists.contains("Gauss"));

computer_scientists.erase("Gauss");
CHECK(ranges::distance(computer_scientists) == 6);
CHECK(not computer_scientists.contains("Gauss"));
```

Inserting an element

```
computer_scientists.insert("Gauss");
CHECK(ranges::distance(computer_scientists) == 7);
CHECK(computer_scientists.contains("Gauss"));

computer_scientists.erase("Gauss");
CHECK(ranges::distance(computer_scientists) == 6);
CHECK(not computer_scientists.contains("Gauss"));
```

Inserting an element

```
computer_scientists.insert("Gauss");
CHECK(ranges::distance(computer_scientists) == 7);
CHECK(computer_scientists.contains("Gauss"));

computer_scientists.erase("Gauss");
CHECK(ranges::distance(computer_scientists) == 6);
CHECK(not computer_scientists.contains("Gauss"));
```

Finding an element

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

CHECK(*ada == "Lovelace");
```

An empty set

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

auto const no_names = absl::flat_hash_set<std::string>{};
REQUIRE(no_names.empty());

CHECK(computer_scientists == no_names);
```

An empty set

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

auto const no_names = absl::flat_hash_set<std::string>{};
REQUIRE(no_names.empty());

CHECK(computer_scientists == no_names);
```

An empty set

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

auto const no_names = absl::flat_hash_set<std::string>{};
REQUIRE(no_names.empty());

CHECK(computer_scientists == no_names);
```

Hash maps

```
auto country_codes = absl::flat_hash_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"},
};
```

Hash maps

```
auto country_codes = absl::flat_hash_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
```

Hash maps

```
auto country_codes = absl::flat_hash_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 maps

```
auto check_code_mapping(
  absl::flat_hash_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);
}
```

Hash maps

```
auto check_code_mapping(
  absl::flat_hash_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);
}
```

Hash maps

```
auto check_code_mapping(
  absl::flat_hash_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);
}
```

Hash maps

```cpp
auto check_code_mapping(
  absl::flat_hash_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);
}
```

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

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

```cpp
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() { return 1; }

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

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

```cpp
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() { return 1; }

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

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

```cpp
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() { return 1; }

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

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

```cpp
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() { return 1; }

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

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

```cpp
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() { return 1; }

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

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

```cpp
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() { return 1; }

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

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

```cpp
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() { return 1; }

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

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

```cpp
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() { return 1; }

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

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

```cpp
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() { return 1; }

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

Program errors

There are 4 types of program errors that we will discuss

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

Compile-time Errors

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

Link-time Errors

```
#include "catch2/catch.hpp"

auto is_cs6771() -> bool;

TEST_CASE("This is all the code")
  CHECK(is_cs6771()); // Link-time error: is_cs6771 not found.
}
```

Run-time Errors

```
auto const course_name = std::string("");
REQUIRE(not course_name.empty()); // Run-time error
```

Logic Errors

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