Concepts of C++ Programming

winter term 2023/24

PD Dr. Tobias Lasser

Computational Imaging and Inverse Problems (CIIP)
Technical University of Munich

Tweedback today

The Tweedback session ID today is zyvk, the URL is:

`https://tweedback.de/zyvk`

- we have two streams on Zulip (`https://zulip.cit.tum.de`): #CPP and #CPP Homeworks
  - #CPP Homeworks is for everything homework related
  - it also has a GitLab bot announcing upstream events when we push updates to the homework assignments
- you can join the CPP stream group by clicking the C++ emoji in #Stream-Gruppen

Contents

1. C++ basics
   - The absolute basics
   - Types and built-in types
   - Variables and initialization
   - Declarations
   - Scopes and lifetimes
   - Namespaces

Finding help: the C++ reference

The basics of C++ are rather simple, just like almost any other programming language

- most of the complexity comes from many special cases and exceptions to many of the rules

We will not be covering every special case / exception!

- the lecture content might be inaccurate or incomplete at times
- you can find an accurate and complete reference documentation of C++ here:

`https://en.cppreference.com/w/cpp`

make sure to use it!

Minimal C++ program

The smallest C++ program:

```cpp
int main() {
}
```

- every C++ program must contain the special function `main()`
- program starts with `main()` (except for the exceptions...)
- program ends when `main()` ends (except for exceptions...)

The four elementary building blocks

Computable algorithms require four elementary building blocks:

- elementary processing step
- sequence of steps
- conditional processing step
- loop

The four elementary building blocks (cont.)

- elementary processing step:

```
int x{0};  // variable definition and initialization
```

- sequence of steps:

```
int x;     // sequence of steps marked by ';'
x = 0;
```

- conditional processing step:

```
if (x == 0)      // test condition in brackets (...)
    x = 2;       // executed if condition is true
else
    x = 0;       // executed if condition is false
```

- loop:

```
while (x > 0) { // execute body while condition true
    --x;        // body is a block enclosed in { }
}
```

Blocks (or scopes) in C++

- blocks are marked using curly brackets in C++:

```cpp
if (x == 0)   // if there is only one statement
    x = 2;    // no block markings are necessary
else
    x = 0;
```

```cpp
if (x == 0) { // but you can use blocks if you want
    x = 2;
}
else {        // you will need them if you want to
    x = 0;    // group several statements
    x -= 1;
}
```

- each block marks a scope (see variable lifetime later)

Loops

- variants of basic while loop:

```
int a{6};
while (a > 0) {
    std::println("{}", a);
    --a;
}
```

```
int a{6};
do {
    std::println("{}", a);
    --a;
} while (a > 0);
```

```
for (int a{6}; a > 0; --a) {
    std::println("{}", a);
}
```

- these variants are functionally equivalent
  - however, in the `for` loop, the variable a is local to the for loop!

Loop control

- loops can be exited early using `break`
- can skip rest of one loop iteration using `continue`

```
int a{12};

while (a > 0) {
    --a;
    if (a % 2)    // skip rest of loop if a not divisible by 2
        continue;

    std::println("{}", a);

    if (a == 2)   // abort loop early if a == 2
        break;
}
```

Contents

1. C++ basics
   - The absolute basics
   - Types and built-in types
   - Variables and initialization
   - Declarations
   - Scopes and lifetimes
   - Namespaces

Type safety

Ideal: a language has type safety

- meaning: every object will be used only according to its type
  - an object will be only used after initialization
  - only operations defined for the object's type will be applied
  - every operation leaves the object in a valid state

- static type safety
  - a program that violates type safety will not compile
  - the compiler (ideally) reports every violation
- dynamic type safety
  - if a program violates type safety it will be detected at run-time
  - a run-time system (ideally) detects every violation

Type safety (cont.)

C++ is neither statically nor dynamically type safe!

- it would interfere with being able to express ideas in code
- it would interfere with the performance goals

But: type safety is very important!

- try very hard not to violate it
- use the available tools:
    - most importantly, the compiler (and its warnings!)
    - static analysis tools (e.g. clang-tidy)
    - dynamic analysis tools (e.g. sanitizers)

C++ is a strongly typed language

- all objects in C++ require a type
- available types:
  - built-in types (e.g. int, float)
  - user-defined types (e.g. classes)

- declaration (and definition) anywhere within a scope (block)

```
TYPE variablename;
```

- initialization with curly brackets

```
TYPE variablename{initializer};
```

(more on this later)

Numbers

- built-in types to support:
  - natural numbers: unsigned using binary representation
  - whole numbers: signed using 2-complement representation (only required since C++20)
  - floating point numbers: using IEEE-754 standard

- all the regular operations are supported:
  - arithmetic: +, -, *, /, %
  - increment/decrement operators: ++, --
  - comparison: <, <=, >, >=, ==, !=
  - logical: && (and), || (or), ! (not)

Natural and whole numbers

<table>
<tr><th>Type</th><th>Equivalent type</th><th>guaranteed size</th><th>size in
32bit system</th><th>size in
64 bit system</th></tr>
<tr><td>short, short int
signed short, signed short int</td><td>short int</td><td rowspan="2">>= 16 bit</td><td rowspan="2">16 bit</td><td rowspan="2">16 bit</td></tr>
<tr><td>unsigned short
unsigned short int</td><td>unsigned short int</td></tr>
<tr><td>int
signed int, signed</td><td>int</td><td rowspan="2">>= 16 bit</td><td rowspan="2">32 bit</td><td rowspan="2">32 bit</td></tr>
<tr><td>unsigned
unsigned int</td><td>unsigned int</td></tr>
<tr><td>long, long int
signed long, signed long int</td><td>long int</td><td rowspan="2">>= 32 bit</td><td rowspan="2">32 bit</td><td rowspan="2">64 bit
(Windows: 32 bit)</td></tr>
<tr><td>unsigned long
unsigned long int</td><td>unsigned long int</td></tr>
<tr><td>long long, long long int
signed long long, signed long long int</td><td>long long int</td><td rowspan="2">>= 64 bit</td><td rowspan="2">64 bit</td><td rowspan="2">64 bit</td></tr>
<tr><td>unsigned long long
unsigned long long int</td><td>unsigned long long int</td></tr>
</table>

Fixed-width integer types

If you need guaranteed sizes, use the types defined `<cstdint>`:

- signed integers: `int8_t`, `int16_t`, `int32_t`, `int64_t` with width of 8, 16, 32, or 64 bits
- unsigned integers: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` with width of 8, 16, 32, or 64 bits
- note: these types are all defined in namespace `std`

Caveat: these types are only defined if the C++ compiler directly supports the type!

```cpp
#include <cstdint>

long a;          // may be 32 or 64 bits
std::int32_t b; // guaranteed to be 32 bits
std::int64_t c; // guaranteed to be 64 bits
```

Integer literals

Integer literals represent constant values

- decimal: `42`
- octal (base 8): `052` (with `0` prefix)
- hexadecimal (base 16): `0x2a` or `0X42` (with `0x` or `0X` prefix)

Careful about sizes: `0xffff` might be -1 or 65535, depending on the type of integer

- suffixes allow to specify the type
  - `unsigned` suffix: `42u` or `42U`
  - `long` suffix: `42l` or `42L`
  - `long long` suffix: `42ll` or `42LL`
  - combinations are possible, e.g. `42ul`, `42ull`

Single quotes can be inserted between digits as separator

- e.g. `1'000'000'000'000ull`

`std::size_t` and `std::ptrdiff_t`

- many size-related C++ standard library functions return an unsigned integer type `std::size_t`
- `std::ptrdiff_t` is, essentially, the signed counterpart of `std::size_t`
- the size of both is implementation-defined!

- since C++23, both types have suffixes for type deduction:
  - `std::ptrdiff_t` suffix: `0z` or `0Z`
  - `std::size_t` suffix: `0uz` or `0UZ`
- this is really useful for `for` loops with `auto` loop counters, see later

Logical values

Logical values are of type `bool`

- possible values are `true` and `false`
- size of `bool` is implementation defined
- often obtained from implicit automatic type conversion (see later)

```
bool condition{true};
// ...
if (condition) {
    // ...
}
```

Floating point numbers

Floating point numbers (usually) in IEEE-754 format

- `float`: 32 bit floating point (approx. 7 decimal digits)
- `double`: 64 bit floating point (approx. 15 decimal digits)
- `long double`: extended precision floating point, usually between 64 and 128 bit
  - 80 bit on x86/x64 architecture (approx. 19 decimal digits), not necessarily IEEE-754 compliant

Floating point types may take special values

- infinity or -infinity (inf or -inf)
- not-a-number (nan)
- negative zero (-0.0)

Floating point literals

Floating point literals represent constant values

- without exponent: `3.141592`, `.5`
- with exponent: `1e9`, `6.26e-34`

By default, a floating point literal is of type `double`

- suffixes allow to specify the type
  - `float` suffix: `1.0f` or `1.0F`
  - `long double` suffix: `1.0l` or `1.0L`

Single quotes can be inserted between digits as separator

- e.g. `6.626'070e-34`

Fixed-width floating point types

Since C++23, fixed-width floating point types are available in `<stdfloat>`

- `float16_t`, `float32_t`, `float64_t`, `float128_t` for 16, 32, 64, or 128 bit IEEE-754 floats
- `bfloat16_t` for a 16 bit "brain" float
- note: these types are all defined in namespace `std`

Caveats:

- these types are *not* aliases of `float`, `double`, etc.
- these types are only defined if the C++ compiler directly supports the type!
- corresponding literal suffixes `f16`, `f32`, etc. are available if the type is available

Characters and strings

Characters and strings are messy in C++, unfortunately.

- character data types:
  - `char`: guaranteed minimum width 8 bit
    - depending on implementation, might be `signed char` or `unsigned char`
    - best to rely only on English alphabetic characters, digits, and basic punctuation characters
  - `char16_t` and `char32_t` to represent UTF-16 and UTF-32 characters
  - `char8_t` to represent UTF-8 characters (since C++20)
  - `wchar_t` is implementation defined...
- string data types: usually represented using the class `std::string`, see later

Character literals

Character literals represent constant values

- any regular character, e.g. `'a'`, `'0'`
- escape sequences, e.g. `'\''`, `'\\'`, `'\n'`, `'\u1234'`

By default, character literals are of type `char`

- prefixes allow to specify the type
  - UTF-8 literal: `u8'a'` for type `char8_t` (since C++20)
  - UTF-16 literal: `u'a'` for type `char16_t`
  - UTF-32 literal: `U'a'` for type `char32_t`

Void type

The type `void` has no values

- no objects of type `void` are allowed
- mainly used to indicate return type for functions without a return value

```
void object;             // ERROR: object of type void
void someFunction() {  // ok: void return type
    // ...
}
```

Contents

1 C++ basics
The absolute basics
Types and built-in types
Variables and initialization
Declarations
Scopes and lifetimes
Namespaces

Variables

Variables always have to be defined before use.

- declaration (and definition): type specifier followed by comma-separated list of declarators (variable names)

```
int a, b;
float bigNumber;
```

- optional: initialization in declaration

```
int a{42};
```

Initialization

There are two kinds of initialization:

- safe: `variableName{<expression>}`
- unsafe: `variableName = <expression>` or `variableName(<expression>)`

Why is one safe and the other unsafe?

- the unsafe version may do (silent) implicit conversions
- the safe version will yield a compiler error (or warning in gcc) if an implicit conversion potentially results in loss of information

Initialization is (unfortunately) optional

- non-local variables are default-initialized (usually zero for built-in types)
- local variables are usually not default-initialized
  - this can lead to undefined behavior when accessing an uninitialized variable

Initialization (cont.)

Examples:

```
double a{3.1415926};
float b = 42;
unsigned c = a;  // compiles, c == 3
unsigned d(b);   // compiles, d == 42
unsigned e{a};   // ERROR: potential information loss
unsigned f{b};   // ERROR: potential information loss
```

Initializers may be arbitrarily complex:

```
double pi{3.1415926}, z{0.30}, a{0.5};
double volume{pi * z * z * a};
```

Contents

1. C++ basics
   - The absolute basics
   - Types and built-in types
   - Variables and initialization
   - Declarations
   - Scopes and lifetimes
   - Namespaces

Declarations

Any name / identifier has to be declared before use, so the compiler knows what entity this name refers to, for example:

- objects, such as local or global variables
- references or pointers to objects
- functions, templates, types, aliases

A declaration takes the following (simplified) form:

- optional prefix (e.g. `static`, `virtual`)
- base type (e.g. `std::vector<double>`, `const int`)
- declarator with optional name (e.g. `n`, `*(*)[]`)
- optional suffix (e.g. `const`, `noexcept`)
- optional initializer / function body (e.g. `{3}`, `{ return x; }`)

Const qualifier

Any type `T` in C++ can be const-qualified:

- either before type: `const T`
- or after type: `T const` (less popular)
- a `const` object is considered immutable and cannot be modified

Example:

```cpp
const int i{1};    // declare and define i, initialized to 1
i = 2;             // ERROR: i is constant
```

Declarations: examples

```cpp
int i;
```

- declares and defines `i` as type `int`, no initialization

```cpp
auto count = 1;
```

- declares and defines `count`, type is automatically deduced as `int`, initialized to 1

```cpp
const float pi{3.1415926};
```

- declares and defines `pi` as type `const float`, initialized to 3.1415926, cannot be modified

```cpp
std::vector<std::string> people{"Martin", "Markus"};
```

- declares and defines `people` as type `std::vector<std::string>`, initialized to contain the strings `"Martin"` and `"Markus"`

```cpp
extern int errorNumber;
```

- declares `errorNumber` as type `int`, no definition!

Declarations: examples (cont.)

```cpp
struct Date { int d, m, y; };
```

- declares `Date` as a structure containing three members

```cpp
int day(Date p) { return p.d; }
```

- declares and defines the function `day`, taking a `Date` as argument, returning an `int`

```cpp
float sqrt(float);
```

- declares the function `sqrt`, taking a `float` as argument, returning a `float`, no definition!

```cpp
struct User;
```

- declares `User` as a structure (forward declaration)

```cpp
using Cmplx = std::complex<double>;
```

- declares and defines an alias `Cmplx` for type `std::complex<double>`

Names

- names are a sequence of letters and digits
- the first character must be a letter
  - the underscore _ is considered a letter
  - non-local names starting with underscore are reserved
- some names (keywords) are reserved
  - for example the built-in types (`int`, `float`, etc.)
  - other examples are `if`, `else`, `switch`, `auto`, `true`, etc.
- names are case sensitive!
  - for example, `Count` and `count` are different names!
- good names are important (and hard to choose)
  - do not encode the type in the name
  - maintain a consistent naming style (e.g. camelCase)
    - use tools to enforce it (e.g. clang-format)

Type aliases

- you can assign new names (aliases) to existing types with the `using` statement:

```cpp
using myInt = long;
using myVector = std::vector<int>;
```

- this is mostly useful for longer types (e.g. templates, see later)

- an older version of this exists too:

```cpp
typedef int myNewInt; // equivalent to using myNewInt = int;
```

Contents

1. C++ basics
   - The absolute basics
   - Types and built-in types
   - Variables and initialization
   - Declarations
   - Scopes and lifetimes
   - Namespaces

Scopes

A scope (or block) is marked by curly brackets: `{}`

- a declaration introduces a name into a scope
- the name can only be used within that scope

There are several types of scopes:

- local scope: corresponding to blocks marked by `{}`
- class scope: the scope inside a class declaration, names are called *member names* or *class member names* (more later)
- namespace scope: the scope inside a namespace declaration, names are called *namespace member names* (more later)
- statement scope: anything declared in the `()` part of a `for`, `while`, `if`, `switch` statement; extends until end of statement
- global scope: basically anything outside the other scopes

Scopes: examples

```
int x{0};          // global x

void f1(int y) { // y local to function f1
    int x;         // local x that hides global x
    x = 1;         // assign to local x
    {
        int x;     // hides the first local x
        x = 2;     // assign to second local x
    }
    x = 3;         // assign to first local x
    y = 2;         // assign to local y
    ::x = 4;       // assign to global x
}

int y = x;         // assigns global x to y (also global)
```

- `::` is the scope resolution operator, so you can refer to hidden global names
- local hidden names cannot be referred to

Scopes: nasty examples

```cpp
int x{42};          // global x

void f2() {
    int x = x;      // nasty! initialize local x with its own value
                    // which is uninitialized...
}

void f3() {         // nasty function!
    int y = x;      // local y initialized with global x == 42
    int x = 21;     // local x hiding global x
    y = x;          // local y assigned local x == 21
}

void f4(int x) {  // x local to function f4, hiding global x
    int x;          // ERROR: x already defined!
}

void f5() {         // no name clash due to statement scope!
    for (int i = 0; i < 5; ++i) std::cout << i << "\n";
    for (auto i : {0, 1, 2, 3, 4}) std::cout << i << "\n";
}
```

Lifetimes of objects

The lifetime of an object

- starts when its constructor completes
- ends when its destructor starts executing

- objects of types without a declared constructor (e.g. `int`) can be considered to have default constructors and destructors that do nothing
- using an object outside its lifetime leads to undefined behavior!

Each object also has a storage duration

- which begins when its memory is allocated and ends when its memory is deallocated
- the lifetime of an object never exceeds its storage duration

Storage durations

Each object has one of the following storage durations:

- automatic: allocated at beginning of scope, deallocated when it goes out of scope
  - e.g. local variables, typically allocated on stack
- static: allocated when program starts, lives until end of program
  - global variables, or variables marked with `static`
- dynamic: allocation and deallocation is handled manually (see later)
  - using `new` and `delete`
  - forgetting deallocation leads to memory leaks! (see later)
  - using an object after deallocation is undefined behavior! (see later)
- thread-local: allocated when thread starts, deallocated automatically when thread ends; each thread gets its own copy
  - declared using `thread_local` keyword

Storage durations: examples

```cpp
int foo{1};                // static storage duration
static int bar{2};         // static storage duration
thread_local int duh{3}; // thread-local storage duration

void f() {
    int x{4};              // automatic storage duration
    static int y{5};       // static storage duration
}
```

Contents

1 C++ basics
- The absolute basics
- Types and built-in types
- Variables and initialization
- Declarations
- Scopes and lifetimes
- Namespaces

Namespaces

Large projects contain many names (variables, functions, classes)

- namespaces allow grouping into logical units
- helps to avoid name clashes

Example:

```
namespace myName {
    int a;
}
```

- a namespace also provides a named scope
- members can be referred to using `myName::`, i.e. a full qualifier

Namespaces: example

```cpp
namespace A {
    void foo() { /* ... */ }
    void bar() {
        foo();    // refers to A::foo
    }
}

namespace B {
    void foo() { /* ... */ }
}

int main() {
    A::foo();    // calls foo() from namespace A
    B::foo();    // calls foo() from namespace B

    foo();       // ERROR: no foo() declared in this scope
}
```

Nested namespaces

Namespaces can be nested:

```
namespace A {
    namespace B {
        void foo() { /* ... */ }
    }
}

namespace A::B {
    void bar() {
        foo();      // refers to A::B::foo
    }
}

int main() {
    A::B::foo();
}
```

Please note: namespaces are open, i.e. you can add names from several namespace declarations (as done above)

Namespaces: practical note

Code is complex and can contain many braces

- ensure readability using comments

```cpp
// ----------------------------
namespace A {
    void foo() {
        // something
    }

    void bar() {
        // something else
    }

} // end namespace A

// ----------------------------
namespace B {
    // more things

} // end namespace B
```

Using namespaces

- fully qualified names are good for readability
- but sometimes it can be tedious or undesired
- the statement `using X::a;` imports the symbol a into the current scope
- the statement `using namespace X` imports all symbols from namespace X into the current scope
    - be careful with this and use sparingly!

```cpp
namespace A { int x; }
namespace B { int y; int z; }
using namespace A;
using B::y;

int main() {
    x = 1;    // refers to A::x
    y = 2;    // refers to B::y
    z = 3;    // ERROR: z undefined
    B::z = 3; // OK
}
```

Summary

1. C++ basics
    - The absolute basics
    - Types and built-in types
    - Variables and initialization
    - Declarations
    - Scopes and lifetimes
    - Namespaces