Concepts of C++ Programming

Lecture 2: Basic Syntax and Object Model

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

Reminder: C++ Reference

These slides will necessarily be inaccurate or incomplete at times.

Use the reference!

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

Comments[5]

- "C-style" or "multi-line" comments: `/*comment */`
- "C++-style" or "single-line" comments: `//comment`

Example:

```
/* This comment is unnecessarily
   split over two lines */
int a = 42;

// This comment is also split
// over two lines
int b = 123;
```

[5] https://en.cppreference.com/w/cpp/comment

Fundamental Types[6]

- `void` – empty type, has no values
  - E.g., used to indicate functions that return no value
- Integer types
  - Boolean type: `bool` (1-bit integer, `true`/`false`)
  - Integer types: `int`, `long`, `unsigned long`, ...
  - Character types: `char`, `char16_t`, . . .
- Floating-point types
  - `float`, `double`, `long double`

[6] https://en.cppreference.com/w/cpp/language/types

Integer Types

- Sign modifiers: `signed` (default), `unsigned`
- Size modifiers: `short`, `long` (≥32 bit), `long long` (≥64 bit)
- Keyword: `int` (optional if modifiers are present)

- Order of keywords is arbitrary
  - `unsigned long long` = `long unsigned int long`
- Signed integers use two's complement (since C++20)

Integer Types: Minimum Width

Canonical Type Specifier	Minimum Width	Minimum Range
`short`	16 bit	-2^{15} to $2^{15}-1$
`unsigned short`		0 to $2^{16}-1$
`int`	16 bit	-2^{15} to $2^{15}-1$
`unsigned`		0 to $2^{16}-1$
`long`	32 bit	-2^{31} to $2^{31}-1$
`unsigned long`		0 to $2^{32}-1$
`long long`	64 bit	-2^{63} to $2^{63}-1$
`unsigned long long`		0 to $2^{64}-1$

- Exact width of integer types is **not** specified by the standard!

Fixed-Width Integer Types[7]

- Use fixed-width types from when... a fixed width is required
- `#include <cstdint>`
- `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
- But: optional, only available if supported by implementation
- Guideline: use fixed-width types only when really required
  - E.g., data structures where size is important, bitwise operations
  - Otherwise, prefer regular integers

[7] https://en.cppreference.com/w/cpp/types/integer

Integer Literals[8]

- Decimal (`42`), octal (`052`), hexadecimal (`0x2a`), binary (`0b101010`)

- `unsigned` suffix: `42u` or `42U`
- `long` suffix: `42l` or `42L`; `long long` suffix: `42ll` or `42LL`
- Both suffixes can be combined, e.g. `42ul`, `42ull`

- Separable by single quotes, e.g. `1’000’000’000ull`, `0b0010’1010`

Quiz: What is the type of the integer literal `0xdeadcabel`?

(Assume 32-bit `int`, 32-bit `long`, as on, e.g., Windows)

A. `int` B. `long` C. `unsigned long` D. `long long`

[8] https://en.cppreference.com/w/cpp/language/integer_literal

Character Types

- Represent character codes and integers
- `signed char`, `unsigned char`
- `char` — implementation-defined whether signed/unsigned!
    - Use `char` only for actual characters, not for arithmetic

- Size: defined as 1 byte
- Size of byte: **at least** 8 bit[9]

- For UTF characters: `char8_t` (C++20), `char16_t`, `char32_t`

[9] Might change for C++26 to exactly 8 bits; proposal: `https://wg21.link/p3477r0`

Character Literals[10]

- E.g. `'a'`, `'b'`, `'€'`
  - Any character from the source character set except: `'`, `\`, newline
- Escape sequences, e.g. `'\''`, `'\\'`, `'\n'`, `'\u1234'`

- UTF-8 prefix: `u8'a'`, `u8'b'`
- UTF-16 prefix: `u'a'`, `u'b'`
- UTF-32 prefix: `U'a'`, `U'b'`

[10] https://en.cppreference.com/w/cpp/language/character_literal

Floating-Point Types

- `float` – usually IEEE-754 32-bit binary format
- `double` – usually IEEE-754 64-bit binary format

- `long double` – extended precision, format varies strongly
  - Some platforms use 64-bit (like `double`), e.g. MSVC on x86
  - Some platforms use 128-bit, e.g. usually AArch64
    (this is typically a softfloat implementation ⇝ slow)
  - On x86, typically 80-bit x87 binary floating-point

- Usual caveats of FP arithmetic apply: infinity, signed zero, NaN

Floating-Point Literals[11]

- Without exponent: `3.1415926`, `.5`
- With exponent: `1e9`, `3.2e20`, `.5e-6`

- `float` suffix: `1.0f` or `1.0F`
- `long double` suffix: `42.0l` or `42.0L`

- Separable by single quotes, e.g. `1’000.000’001`, `.141’592e12`

[11] https://en.cppreference.com/w/cpp/language/floating_literal

Operator Precedence Table (1)[12]

Prec.	Operator	Description	Associativity
1	`::`	Scope resolution	left-to-right
2	`a++   a--`	Postfix increment/decrement	left-to-right
	`<type>()   <type>{}`	Functional Cast	
	`a()   a[]`	Function Call/Subscript	
	`.   ->`	Member Access	
3	`++a   --a`	Prefix increment/decrement	right-to-left
	`+a   -a   !a   ~a`	plus/minus/logical not/bitwise not	
	`(<type>)`	C-style cast	
	`*a   &a`	Dereference/Address-of	
	`sizeof`	Size-of	
	`new   new[]`	Dynamic memory allocation	
	`delete   delete[]`	Dynamic memory deallocation	

[12] https://en.cppreference.com/w/cpp/language/operator_precedence

Operator Precedence Table (2)

Prec.	Operator	Description	Associativity
4	`.*` `->*`	Pointer-to-member	left-to-right
5	`a*b` `a/b` `a%b`	Multiplication/Division/Remainder	left-to-right
6	`a+b` `a-b`	Addition/Subtraction	left-to-right
7	`<<` `>>`	Bitwise shift	left-to-right
8	`<=>`	Three-way comparison	left-to-right

9	`<` `<=`
	`>` `>=`
		Relational $<$ and $\leq$
		Relational $>$ and $\geq$
			left-to-right

10	`==` `!=`	Relational $=$ and $\neq$	left-to-right

Operator Precedence Table (3)

Prec.	Operator	Description	Associativity
11	`&`	Bitwise AND	left-to-right
12	`^`	Bitwise XOR	left-to-right
13	`|`	Bitwise OR	left-to-right
14	`&&`	Logical AND	left-to-right
15	`||`	Logical OR	left-to-right

16	`a?b:c`
	`throw`
	`=`
	`+=  -=  *=  /=  %=`
	`<<= >>= &=  ^=  |=`
		Ternary conditional
		throw operator
		Direct assignment
		Compound assignment
		Compound assignment
			right-to-left

17	`,`	Comma	left-to-right

Observable Behavior

Observable behavior of C++ programs precisely defined, unless:

- *implementation-defined behavior* – documented by C++ implementation
- *unspecified behavior* – one of multiple options can happen
  - E.g., evaluation order of function arguments: one permutation must happen

- program *ill-formed* – syntax/semantic error, compiler must diagnose
- program *ill-formed, no diagnostic required* – semantically invalid, hard to diagnose
  - Typically not detectable during compilation, not too many cases

- *undefined behavior* – the standard imposes no requirements

Undefined Behavior[14] (UB)

- Some violations of language rules are undefined behavior: standard enforces no restrictions ⇝ **anything** can happen
  - Typically cases, where checks would be costly or impossible

⇒ A C++ program **must never** contain undefined behavior!

- Examples: out-of-bounds array access, signed integer overflow, shift by negative index, shift larger than value size, . . .
  - Signed integers: UB on overflow; unsigned integers: well-defined wrap

- Compiler can assume that program contains no undefined behavior[13]
  - Allows for more optimizations, e.g. eliminate some checks

[13] https://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

[14] https://en.cppreference.com/w/cpp/language/ub

Undefined Behavior – Example

Quiz: Which answer is correct?

```
bool f1(int x) { return x + 1 > x; }
bool f2(unsigned x) { return x + 1 > x; }
```

A. The return value of `f1` is always `false`.

B. The return value of `f2` is always `true`.

C. The return value of `f1` depends on the parameter.

D. The return value of `f2` depends on the parameter.

E. `f2` might invoke undefined behavior.

Variables[15]

- Declaration: type specifier followed by declarators (variable names)

- Declarator can optionally be followed by an initializer
- No initializer: *default-initialized*
  - Non-local variables: zero-initialized
  - Local variables: **not initialized**
- Access of uninitialized variable is **undefined behavior**

```
void foo() {
  unsigned i = 0, j;
  unsigned meaningOfLife = 42;
}
```

[15] https://en.cppreference.com/w/cpp/language/declarations

Variable Initializers[16]

- `variableName(<expression>)`
- `variableName = <expression>`
- `variableName{<expression>}` (error on possible information loss)

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

[16] https://en.cppreference.com/w/cpp/language/initialization

Simple Statements[17]

Declaration statement: Declaration followed by a semicolon

```
int i = 0;
```

Expression statement: Any expression followed by a semicolon

```
i + 5; // valid, but useless
foo(); // valid and possibly useful
```

Compound statement (blocks): Brace-enclosed sequence of statements

```
{ // start of block
    int i = 0; // declaration statement
} // end of block, i goes out of scope
int i = 1; // declaration statement
```

[17] https://en.cppreference.com/w/cpp/language/statements

Scope[18]

Names in a C++ program are valid only within their *scope*

- The scope of a name begins at its point of declaration
- The scope of a name ends at the end of the relevant block
- Scopes may be shadowed resulting in discontiguous scopes (bad practice)

```
int a = 21;
int b = 0;
{
  int a = 1; // scope of the first a is interrupted
  int c = 2;
  b = a + c + 39; // a refers to the second a, b == 42
} // scope of the second a and c ends
b = a; // a refers to the first a, b == 21
b += c; // ERROR: c is not in scope
```

[18] https://en.cppreference.com/w/cpp/language/scope

If Statement[19]

- Conditionally execute another statement
- Condition converted to `bool` decides which branch is taken
- Optional initialization statement
- Optional `else` branch

```
if (value < 42)
  valueLessThan42();
else
  valueTooLarge();
```

```
if (unsigned n = compute(); n > 4) {
  // do something
}
// The latter is equivalent to:
{
  unsigned n = compute();
  if (n > 4) {
    // do something
  }
}
```

[19] https://en.cppreference.com/w/cpp/language/if

If Statement Nesting

- `else` is associated with the closest `if` that has no `else`

```
// INTENTIONALLY BUGGY!
if (condition0)
  if (condition1)
    // do something if (condition0 && condition1) == true
else
  // do something if condition0 == false
```

- When in doubt, use curly braces to make scopes explicit

```
// Working as intended
if (condition0) {
  if (condition1)
    // do something if (condition0 && condition1) == true
} else {
  // do something if condition0 == false
}
```

Switch Statements[20]

- Conditional control flow transfer based on integral type
- Constant values for `case`, must be unique
- `break` exits `switch`
- Implicit fallthrough!
  - Use `[[fallthrough]];` when intended
- Condition can have declaration

```cpp
switch (compute()) {
case 42:
  // do something for 42
  break;
case 20:
  // do something for 20
  [[fallthrough]];
case 21:
case 22:
  // do something for 20/21/22
  break;
default:
  break;
}
```

[20] https://en.cppreference.com/w/cpp/language/switch

While and Do-While Loops

- while:[21] repeatedly execute statement while condition is true

```
unsigned i = 42;
while (i < 42) {
  // never executed
}
```

- do–while:[22] like while, but execute body at least once

```
unsigned i = 42;
do {
  // executed once
} while (i < 42);
```

- break/continue to exit loop/skip remainder of body

[21] https://en.cppreference.com/w/cpp/language/while

[22] https://en.cppreference.com/w/cpp/language/do

For Loops[23]

```
for (unsigned i = 0; i < 10; ++i) {
  // iterate 0, 1, 2, ..., 9
}
for (unsigned i = 0, len = getLength(); i != len; ++i) {
  // do something; doesn’t call getLength() every iteration
}
for (unsigned i = 42; i-- > 0; ) {
  // iterate 41, 40, ..., 0
}
uint8_t i = 0;
for (; i < 256; ++i)
  std::println("{}", i); // hmmm....
```

Quiz: What could be a problem of the last loop?

A. No Problem B. Syntax Error C. Endless Loop D. Undefined Behavior

[23] https://en.cppreference.com/w/cpp/language/for

Basic Functions[24]

- Associate a sequence of statements (body) with a name
- Function can have parameters and a return type (can be `void`)
- Non-`void` functions must execute `return` statement
- Arguments are passed **by value** (unlike Java for classes)
  - Pass-by-reference requires explicit annotation, see later

```
void procedure(unsigned parameter0, double parameter1) {
  // do something with parameter0 and parameter1
}
unsigned meaningOfLife() {
  // complex computation, takes 7.5 million years
  return 42;
}
```

[24] https://en.cppreference.com/w/cpp/language/function

Basic Function Arguments

- Parameters can be unnamed ⇝ unusable, but still required on call
- Function can specify default arguments[25] in parameter list
  - After first param with default value, all must have a default value

```
unsigned meaningOfLife(unsigned /*unused*/) {
  return 42;
}
unsigned addNumbers(int a, int b = 2, int c = 3) {
  unsigned v = meaningOfLife(); // ERROR: expected argument
  unsigned w = meaningOfLife(123); // OK
  return a + b + c;
}
int main() {
  int x = addNumbers(1); // x == 6
  int y = addNumbers(1, 1); // y == 5
  int z = addNumbers(1, 1, 1); // z == 3
}
```

[25] https://en.cppreference.com/w/cpp/language/default_arguments

Namespaces[26]

- Large projects contain many names ⇝ organize in logical units
- *namespaces* allow preventing name conflicts

```
namespace A {
void foo() { /* do something */ }
void bar() { foo(); /* refers to A::foo */ }
} // end namespace A
namespace B {
void foo() { /* do something */ }
} // end namespace B
int main() {
  A::foo(); // qualified name lookup
  B::foo(); // qualified name lookup
  foo(); // ERROR: foo was not declared in this scope
}
```

[26] https://en.cppreference.com/w/cpp/language/namespace

Namespace Nesting

- Namespaces can be nested

```
namespace A {
namespace B {
void foo() { /* do something */ }
} // end namespace B
} // end namespace A

// equivalent definition
namespace A::B {
void bar() { foo(); /* refers to A::B::foo */ }
} // end namespace A::B

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

Namespaces: `using` and Conventions

- Typically: add comments to closing namespace brace

- Always using fully qualified names makes code easier to read
- But: sometimes, source is obvious and typing cumbersome...
  - `using namespace X;` imports *everything* from `X`
  - `using X::a;` imports only *a* from `X`

```
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 was not declared in this scope
  B::z = 3; // OK
}
```

Memory Model

- Fundamental storage unit: *byte*
  - There can (theoretically) be more than 8 bits in a byte
- Memory consists of one or more contiguous sequences of bytes
  - Memory can have holes, e.g. due to virtual memory

- Every byte has a unique address

Objects[27]

- Object: region of storage; properties:
  - Size (see next slides)
  - Alignment (see next slides)
  - Storage duration (see next slides)
  - Lifetime (see next slides)
  - Type
  - Value
  - Optionally: name

- C++ programs create, destroy, refer to, access, and manipulate objects
- Examples for objects: local/global variables, parameters
  - Not objects: functions, references, values

[27] https://en.cppreference.com/w/cpp/language/object

Object Size and Alignment

- Size and alignment requirements are defined by the type

- `sizeof` operator[28]: query size in bytes of object/type
  - `sizeof(char)` = `sizeof(std::byte)` = 1
  - All other sizes implementation-defined

- `alignof` operator[29]: query minimum alignment in bytes of type
  - Depending on implementation, some values must be aligned in memory
  - Alignment is always a power of 2
  - Address must be a multiple of the alignment

[28] https://en.cppreference.com/w/cpp/language/sizeof

[29] https://en.cppreference.com/w/cpp/language/alignof

Storage Duration[31]

- Every object has a storage duration

Storage Duration	Begin	End	Note/Example
automatic	Scope begin	Scope end	Local variables
static	Program begin	Program end	Global variables
thread	Thread start	Thread end	`thread_local` vars
dynamic	`new`	`delete`	

- Static: allocated/initialized before `main` in non-guaranteed order[30]
- Thread: one copy of the object per thread
- Dynamic: allocation/deallocation must be done manually

[30] https://en.cppreference.com/w/cpp/language/siof

[31] https://en.cppreference.com/w/cpp/language/storage_duration

Lifetime[32]

Lifetime of an object...

- starts when it is fully *initialized*
- ends when destructor called (classes)
  or storage is deallocated/reused (others)
- never exceeds the lifetime of the storage (see storage duration)

- Using an object outside its lifetime is **undefined behavior**
- This is a main source of memory bugs

- Compilers can only warn about very basic errors

⇒ If compiler warns, always **fix your program**

[32] https://en.cppreference.com/w/cpp/language/lifetime

Lifetime: Example

Quiz: When does the lifetime of p end?

```
int g;
void matterOfLifeOrDeath(unsigned a) {
  thread_local int t = 1;
  unsigned c = a;
  {
    unsigned p = a + 1;
  }
  unsigned m = t - 1;
}
```

A. At the end of the function.

B. At the end of the innermost block.

C. At the end of the program.

D. When the underlying stack space is reuseed (e.g., for `m`).

Lifetime: Example

Quiz: What is problematic about this function?

```
int fancyZero() { // fancy way to return zero
  int x = x ^ x;
  return x;
}
```

A. Ill-formed/compile error: `x` used before its declaration.

B. Undefined behavior: signed integer overflow.

C. Undefined behavior: `x` used outside its lifetime.

D. Undefined behavior: `x` used outside its storage duration.

Basic Syntax and Object Model – Summary

- Fundamental types: void, integral, floating-point
- Exact width, representation, etc. not specified by standard
- Undefined behavior means anything can happen
- Undefined behavior must therefore never happen
- Basic syntax similar to other C-like languages, with additions
- Use namespaces to avoid naming collisions
- C++ programs resolve around working with objects
- Objects' lifetime is often implicit, leading to subtle bugs

Basic Syntax and Object Model – Questions

- What is the required minimum size of an `unsigned int`?
- Why is arithmetic on `char` problematic?
- Why is `long double` rarely used?
- What can happen when undefined behavior is encountered?
- How can compilers use undefined behavior for optimizations?
- Which variable initializer prevents loss of accuracy?
- What is the storage duration of an object?
- What is the relation between storage duration and lifetime?