

Introduction to C++

Foundations of Data Engineering

Moritz Sichert

Introduction

What is C++?

Multi-paradigm general-purpose programming language

- Imperative programming
- Object-oriented programming
- Generic programming
- Functional programming

Key characteristics

- Compiled language
- Statically typed language
- Facilities for low-level programming

A Brief History of C++

Initial development

- Bjarne Stroustrup at Bell Labs (since 1979)
- In large parts based on C
- Inspirations from Simula67 (classes) and Algol68 (operator overloading)

First ISO standardization in 1998 (C++98)

- Further amendments in following years (C++03, C++11, C++14, C++17)
- Current standard: C++20
- Next standard: C++23

Why Use C++?

Performance

- Flexible level of abstraction (very low-level to very high-level)
- High-performance even for user-defined types
- Direct mapping of hardware capabilities
- Zero-overhead rule: “What you don’t use, you don’t pay for.” (Bjarne Stroustrup)

Flexibility

- Choose suitable programming paradigm
- Comprehensive ecosystem (tool chains & libraries)
- Scales easily to very large systems (with some discipline)
- Interoperability with other programming languages (especially C)

Useful Websites

C++ Reference Wiki:

- URL: <https://en.cppreference.com/w/cpp>
- Best website for up-to-date C++ reference

Compiler Explorer by Matt Godbolt:

- Allows interactive viewing of the assembly generated by various C++ compilers
- We host an instance at <https://compiler.db.in.tum.de/>

Introduction to the C++ Ecosystem

Hello World in C++

myprogram.cpp

```
#include <iostream>
int main(int argc, char** argv) {
    std::cout << "Hello " << argv[1] << "!" << std::endl;
    return 0;
}
```



```
$ g++ -std=c++20 -Wall -Werror -o myprogram ./myprogram.cpp
$ ./myprogram World
Hello World!
```


Generating an Executable Program

- Programs that transform C++ files into executables are called *compilers*
- Popular compilers: gcc (GNU), clang (llvm)
- Minimal example to compile the hello world program with gcc:

```
$ g++ -o myprogram ./myprogram.cpp
```

- Internally, the compiler is divided into:
 - Preprocessor
 - Compiler
 - Linker

Compiler Flags

General syntax to run a compiler: `c++ [flags] -o output inputs...`

Most common flags:

<code>-std=c++20</code>	Set C++ standard version
<code>-O0</code>	no optimization
<code>-O1</code>	optimize a bit, assembly mostly readable
<code>-O2</code>	optimize more, assembly not readable
<code>-O3</code>	optimize most, assembly not readable
<code>-Os</code>	optimize for size, similar to <code>-O3</code>
<code>-Wall</code>	Enable most warnings
<code>-Wextra</code>	Enable warnings not covered by <code>-Wall</code>
<code>-Werror</code>	Treat all warnings as errors
<code>-march=native</code>	Enable optimizations supported by your CPU
<code>-g</code>	Enable debug symbols

make

- C++ projects usually consist of many `.cpp` (*implementation files*) and `.hpp` (*header files*) files
- Each implementation file needs to be compiled into an object file first, then all object files must be linked
- Very repetitive to do this by hand
- When one `.cpp` file changes, only the corresponding object file should be recompiled, not all
- When one `.hpp` file changes, only implementation files that use it should be recompiled
- `make` is a program that can automate this
- Requires a `Makefile`
- GNU `make` manual:
<https://www.gnu.org/software/make/manual/make.html>

CMake

- make prevents writing many repetitive compiler commands
- Still, extra flags must be specified manually (e.g. `-l` to link an external library)
- On different systems the same library may require different flags
- CMake is a tool specialized for C and C++ projects that uses a `CMakeLists.txt` to generate `Makefiles` or files for other build systems (e.g. ninja, Visual Studio)
- Also, the C++ IDE CLion uses CMake internally
- `CMakeLists.txt` consists of a series of *commands*
- CMake Reference Documentation:
<https://cmake.org/cmake/help/latest/>

Basic CMakeLists.txt

————— CMakeLists.txt —————

```
cmake_minimum_required(VERSION 3.10)
project(myprogram)
set(MYPROGRAM_FILES sayhello.cpp saybye.cpp)
add_executable(myprogram myprogram.cpp ${MYPROGRAM_FILES})
```



```
$ mkdir build; cd build # create a separate build directory
$ cmake .. # generate Makefile from CMakeLists.txt
-- The C compiler identification is GNU 8.2.1
-- The CXX compiler identification is GNU 8.2.1
[...]
-- Configuring done
-- Generating done
-- Build files have been written to: /home/X/myproject/build
$ make
Scanning dependencies of target myprogram
[ 25%] Building CXX object CMakeFiles/myprogram.dir/myprogram.cpp.o
[ 50%] Building CXX object CMakeFiles/myprogram.dir/sayhello.cpp.o
[ 75%] Building CXX object CMakeFiles/myprogram.dir/saybye.cpp.o
[100%] Linking CXX executable myprogram
```

Basic C++ Syntax

Overview

Common set of basic features shared by a wide range of programming languages

- Built-in types (integers, characters, floating point numbers, etc.)
- Variables (“names” for entities)
- Expressions and statements to manipulate values of variables
- Control-flow constructs (`if`, `for`, etc.)
- Functions, i.e. units of computation

Supplemented by additional functionality


- Programmer-defined types (`struct`, `class`, etc.)
- Library functions

The C++ Reference Documentation

C++ is in essence a simple language

- Limited number of basic features and rules
- **But:** There is a corner case to most features and an exception to most rules
- **But:** Some features and rules are rather obscure

These slides will necessarily be inaccurate or incomplete at times

- <https://en.cppreference.com/w/cpp> provides an excellent and complete reference documentation of C++
- Every C++ programmer should be able to read and understand the reference documentation
- Slides that directly relate to the reference documentation contain the  symbol with a link to the relevant webpage in the slide header

Look at these links and familiarize yourself with the reference documentation!



Comments

C++ supports two types of comments

- “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;
```



Fundamental Types

C++ defines a set of primitive types

- Void type
- Boolean type
- Integer types
- Character types
- Floating point types

All other types are composed of these fundamental types in some way



Void Type

The void type has no values

- Identified by the C++ keyword `void`
- No objects of type `void` are allowed
- Mainly used as a return type for functions that do not return any value
- Pointers to `void` are also permitted

```
void* pointer;           // OK: pointer to void
void object;            // ERROR: object of type void
void doSomething() {    // OK: void return type
    // do something important
}
```



Boolean Type

The boolean type can hold two values

- Identified by the C++ keyword `bool`
- Represents the truth values `true` and `false`
- Quite frequently obtained from implicit automatic type conversion

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



Integer Type Overview

Overview of the integer types as specified by the C++ standard

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

The exact width of integer types is **not** specified by the standard!



Fixed-Width Integer Types

Sometimes we need integer types with a guaranteed width

- Use fixed-width integer types defined in `<cstdint>` header
- `int8_t`, `int16_t`, `int32_t` and `int64_t` for signed integers of width 8, 16, 32 or 64 bit, respectively
- `uint8_t`, `uint16_t`, `uint32_t` and `uint64_t` for unsigned integers of width 8, 16, 32 or 64 bit, respectively

Only defined if the C++ implementation directly supports the type

```
#include <cstdint>

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

Integer Type Guidelines

Use basic (i.e. non-fixed-width) integer types by default

- They guarantee a minimum range that can be supported
- Most of the time we do not need to know an exact maximum value
- Usually (**unsigned**) **int** or **long** are a reasonable choice

Only use fixed-width integer types where absolutely required

- E.g. in data structures that need to have deterministic fixed size
- E.g. in library calls
- E.g. for bitwise operations that rely on masks, shifts etc.

Do not prematurely optimize for space consumption

- Registers on modern CPUs are likely to be 64 bit wide anyway
- Most of the time a program only becomes susceptible to overflow bugs if narrow integer types are used without good reason



Floating Point Types

Floating point types of varying precision

- `float` usually represents IEEE-754 32 bit floating point numbers
- `double` usually represents IEEE-754 64 bit floating point numbers
- `long double` is a floating point type with extended precision (varying width depending on platform and OS, usually between 64 bit and 128 bit)

Floating point types may support special values

- Infinity
- Negative zero
- Not-a-number



Implicit Conversions (1)

Type conversions may happen automatically

- If we use an object of type A where an object of type B is expected
- Exact conversion rules are highly complex (full details in the reference documentation)

Some common examples

- If one assigns an integral type to `bool` the result is `false` if the integral value is `0` and `true` otherwise
- If one assigns `bool` to an integral type the result is `1` if the value is `true` and `0` otherwise
- If one assigns a floating point type to an integral type the value is truncated
- If one assigns an out-of-range value to an unsigned integral type of width w , the result is the original value modulo 2^w

Implicit Conversions (2)

Example

```
uint16_t i = 257;
uint8_t j = i; // j is 1

if (j) {
    /* executed if j is not zero */
}
```



Undefined Behavior (1)

In some situations the behavior of a program is not well-defined

- E.g. overflow of an unsigned integer is well-defined (see previous slide)
- **But:** Signed integer overflow results in **undefined behavior**
- We will encounter undefined behavior every once in a while

Undefined behavior falls outside the specification of the C++ standard

- The compiler is allowed to do anything when it encounters undefined behavior
- Fall back to some sensible default behavior
- Do nothing
- Print 42
- Do anything else you can think of

A C++ program must never contain undefined behavior!

Undefined Behavior (2)

Example

foo.cpp

```
int foo(int i) {  
    if ((i + 1) > i)  
        return 42;  
  
    return 123;  
}
```

foo.o

```
foo(int):  
    movl    $42, %eax  
    retq
```



Variables

Variables need to be declared before they can be used

- Simple declaration: Type specifier followed by comma-separated list of declarators (variable names) followed by semicolon
- Variable names in a simple declaration may optionally be followed by an initializer

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



Const & Volatile Qualifiers

Any type T in C++ (except function and reference types) can be *cv-qualified*

- const-qualified: `const T`
- volatile-qualified: `volatile T`
- cv-qualifiers can appear in any order, before or after the type

Semantics

- `const` objects cannot be modified
- Any read or write access to a `volatile` object is treated as a visible side effect for the purposes of optimization
- `volatile` should be avoided in most cases (it is likely to be deprecated in future versions of C++)
- Use *atomics* instead



Expression Fundamentals

C++ provides a rich set of operators

- Operators and operands can be composed into expressions
- Most operators can be overloaded for custom types

Fundamental expressions

- Variable names
- Literals

Operators act on a number of operands

- Unary operators: E.g. negation ($-$), address-of ($\&$), dereference ($*$)
- Binary operators: E.g. equality ($==$), multiplication ($*$)
- Ternary operator: $a ? b : c$



Value Categories

Each expression in C++ is characterized by two independent properties

- Its *type* (e.g. `unsigned`, `float`)
- Its *value category*
- Operators may require operands of certain value categories
- Operators result in expressions of certain value categories

Broadly (and inaccurately) there are two value categories: *lvalues* and *rvalues*

- *lvalues* refer to the identity of an object
- *rvalues* refer to the value of an object
- Modifiable *lvalues* can appear on the left-hand side of an assignment
- *lvalues* and *rvalues* can appear on the right-hand side of an assignment

C++ actually has a much more sophisticated taxonomy of expressions

- Relevant for *move* operations



Arithmetic Operators (1)

Operator	Explanation
+a	Unary plus
-a	Unary minus
a + b	Addition
a - b	Subtraction
a * b	Multiplication
a / b	Division
a % b	Modulo
~a	Bitwise NOT
a & b	Bitwise AND
a b	Bitwise OR
a ^ b	Bitwise XOR
a << b	Bitwise left shift
a >> b	Bitwise right shift

C++ arithmetic operators have the usual semantics



Arithmetic Operators (2)

Incorrectly using the arithmetic operators can lead to undefined behavior, e.g.

- Signed overflow
- Division by zero
- Shift by a negative offset
- Shift by an offset larger than the width of the type



Logical and Relational Operators (1)

Operator	Explanation
!a	Logical NOT
a && b	Logical AND (short-circuiting)
a b	Logical OR (short-circuiting)
a == b	Equal to
a != b	Not equal to
a < b	Less than
a > b	Greater than
a <= b	Less than or equal to
a >= b	Greater than or equal to
a <=> b	Three-way comparison

Most C++ logical and relational operators have the usual semantics

Logical and Relational Operators (2)

The three-way comparison (or spaceship) operator is a useful addition in C++20

- $(a <=> b) < 0$ if $a < b$
- $(a <=> b) == 0$ if $a == b$
- $(a <=> b) > 0$ if $a > b$
- Can be generated by the compiler automatically in some cases
- Facilitates, for example, sorting values



Assignment Operators (1)

Operator	Explanation
<code>a = b</code>	Simple assignment
<code>a += b</code>	Addition assignment
<code>a -= b</code>	Subtraction assignment
<code>a *= b</code>	Multiplication assignment
<code>a /= b</code>	Division assignment
<code>a %= b</code>	Modulo assignment
<code>a &= b</code>	Bitwise AND assignment
<code>a = b</code>	Bitwise OR assignment
<code>a ^= b</code>	Bitwise XOR assignment
<code>a <<= b</code>	Bitwise left shift assignment
<code>a >>= b</code>	Bitwise right shift assignment

Notes

- The left-hand side of an assignment operator must be a modifiable lvalue
- For built-in types `a OP= b` is equivalent to `a = a OP b` except that `a` is only evaluated once

Assignment Operators (2)

The assignment operators return a reference to the left-hand side

```
unsigned a, b, c;  
a = b = c = 42; // a, b, and c have value 42
```

Usually rarely used, with one exception

```
unsigned d;  
if (d = computeValue()) {  
    // executed if d is not zero  
} else {  
    // executed if d is zero  
}  
  
// unconditionally do something with d
```



Increment and Decrement Operators

Operator	Explanation
<code>++a</code>	Prefix increment
<code>--a</code>	Prefix decrement
<code>a++</code>	Postfix increment
<code>a--</code>	Postfix decrement

Return value differs between prefix and postfix variants

- Prefix variants increment or decrement the value of an object and return a *reference* to the result
- Postfix variants create a copy of an object, increment or decrement the value of the original object, and return the copy



Ternary Conditional Operator

Operator	Explanation
<code>a ? b : c</code>	Conditional operator

Semantics

- `a` is evaluated and converted to `bool`
- If the result was `true`, `b` is evaluated
- Otherwise `c` is evaluated

The type and value category of the resulting expression depend on the operands

```
int n = (1 > 2) ? 21 : 42; // 1 > 2 is false, i.e. n == 42
int m = 42;
((n == m) ? m : n) = 21; // n == m is true, i.e. m == 21

int k{(n == m) ? 5.0 : 21}; // ERROR: narrowing conversion
((n == m) ? 5 : n) = 21; // ERROR: assigning to rvalue
```




Simple Statements

Declaration statement: Declaration followed by a semicolon

```
int i = 0;
```

Expression statement: Any expression followed by a semicolon

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

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
```



Scope

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 discontinuous 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
```



If Statement (1)

Conditionally executes another statement

```
if (init-statement; condition)
    then-statement
else
    else-statement
```

Explanation

- If *condition* evaluates to **true** after conversion to **bool**, *then-statement* is executed, otherwise *else-statement* is executed
- Both *init-statement* and the else branch can be omitted
- If present, *init-statement* must be an expression or declaration statement
- *condition* must be an expression statement or a single declaration
- *then-statement* and *else-statement* can be arbitrary (compound) statements

If Statement (2)

The *init-statement* form is useful for local variables only needed inside the `if`

```
if (unsigned value = computeValue(); value < 42) {  
    // do something  
} else {  
    // do something else  
}
```

Equivalent formulation

```
{  
    unsigned value = computeValue();  
    if (value < 42) {  
        // do something  
    } else {  
        // do something else  
    }  
}
```

If Statement (3)

In nested `if`-statements, the `else` is associated with the closest `if` that does not have an `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 Statement (1)

Conditionally transfer control to one of several statements

```
switch (init-statement; condition)
    statement
```

Explanation

- *condition* may be an expression or single declaration that is convertible to an enumeration or integral type
- The body of a `switch` statement may contain an arbitrary number of case *constant*: labels and up to one `default`: label
- The constant values for all case: labels must be unique
- If *condition* evaluates to a value for which a case: label is present, control is passed to the labelled statement
- Otherwise, control is passed to the statement labelled with `default`:
- The `break`; statement can be used to exit the `switch`

Switch Statement (2)

Regular example

```
switch (computeValue()) {  
    case 21:  
        // do something if computeValue() was 21  
        break;  
    case 42:  
        // do something if computeValue() was 42  
        break;  
    default:  
        // do something if computeValue() was != 21 and != 42  
        break;  
}
```

Switch Statement (3)

The body is executed sequentially until a `break;` statement is encountered

```
switch (computeValue()) {  
    case 21:  
    case 42:  
        // do something if computeValue() was 21 or 42  
        break;  
    default:  
        // do something if computeValue() was != 21 and != 42  
        break;  
}
```

Compilers may generate warnings when encountering such fall-through behavior

- Use special `[[fallthrough]];` statement to mark intentional fall-through



While Loop

Repeatedly executes a statement

```
while (condition)
    statement
```

Explanation

- Executes *statement* repeatedly until the value of *condition* becomes **false**. The test takes place before each iteration.
- *condition* may be an expression that can be converted to **bool** or a single declaration
- *statement* may be an arbitrary statement
- The **break**; statement may be used to exit the loop
- The **continue**; statement may be used to skip the remainder of the body



Do-While Loop

Repeatedly executes a statement

```
do
    statement
while (condition);
```

Explanation

- Executes *statement* repeatedly until the value of *condition* becomes **false**. The test takes place after each iteration.
- *condition* may be an expression that can be converted to **bool** or a single declaration
- *statement* may be an arbitrary statement
- The **break**; statement may be used to exit the loop
- The **continue**; statement may be used to skip the remainder of the body

While vs. Do-While

The body of a do-while loop is executed at least once

```
unsigned i = 42;

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

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



For Loop (1)

Repeatedly executes a statement

```
for (init-statement; condition; iteration-expression)  
    statement
```

Explanation

- Executes *init-statement* once, then executes *statement* and *iteration-expression* repeatedly until *condition* becomes **false**
- *init-statement* may either be an expression or declaration
- *condition* may either be an expression that can be converted to **bool** or a single declaration
- *iteration-expression* may be an arbitrary expression
- All three of the above statements may be omitted
- The **break**; statement may be used to exit the loop
- The **continue**; statement may be used to skip the remainder of the body

For Loop (2)

```
for (unsigned i = 0; i < 10; ++i) {  
    // do something  
}  
  
for (unsigned i = 0, limit = 10; i != limit; ++i) {  
    // do something  
}
```

Beware of integral overflows (signed overflows are undefined behavior!)

```
for (uint8_t i = 0; i < 256; ++i) {  
    // infinite loop  
}  
  
for (unsigned i = 42; i >= 0; --i) {  
    // infinite loop  
}
```



Basic Functions (1)

Functions in C++

- Associate a sequence of statements (the *function body*) with a name
- Functions may have zero or more *function parameters*
- Functions can be invoked using a function-call expression which initializes the parameters from the provided arguments

Informal function definition syntax

```
return-type name ( parameter-list ) {  
    statement  
}
```

Informal function call syntax

```
name ( argument-list );
```

Basic Functions (2)

Function may have `void` return type

```
void procedure(unsigned parameter0, double parameter1) {  
    // do something with parameter0 and parameter1  
}
```

Functions with non-`void` return type must contain a `return` statement

```
unsigned meaningOfLife() {  
    // extremely complex computation  
    return 42;  
}
```

The `return` statement may be omitted in the main-function of a program (in which case zero is implicitly returned)

```
int main() {  
    // run the program  
}
```

Basic Functions (3)

Function parameters may be unnamed, in which case they cannot be used

```
unsigned meaningOfLife(unsigned /*unused*/) {  
    return 42;  
}
```

An argument must still be supplied when invoking the function

```
unsigned v = meaningOfLife();    // ERROR: expected argument  
unsigned w = meaningOfLife(123); // OK
```


Argument Passing

Argument to a function are passed **by value** in C++

```
unsigned square(unsigned v) {  
    v = v * v;  
    return v;  
}  
  
int main() {  
    unsigned v = 8;  
    unsigned w = square(v); // w == 64, v == 8  
}
```

C++ differs from other programming languages (e.g. Java) in this respect

- Parameters can *explicitly* be passed by reference
- Essential to keep argument-passing semantics in mind, especially when user-defined classes are involved



Default Arguments

A function definition can include default values for some of its parameters

- Indicated by including an initializer for the parameter
- After a parameter with a default value, all subsequent parameters must have default values as well
- Parameters with default values may be omitted when invoking the function

```
int foo(int a, int b = 2, int c = 3) {  
    return a + b + c;  
}
```

```
int main() {  
    int x = foo(1);           // x == 6  
    int y = foo(1, 1);       // y == 5  
    int z = foo(1, 1, 1);    // z == 3  
}
```



Function Overloading (1)

Several functions may have the same name (*overloaded*)

- Overloaded functions must have distinguishable parameter lists
- Calls to overloaded functions are subject to *overload resolution*
- Overload resolution selects which overloaded function is called based on a set of complex rules

Informally, parameter lists are distinguishable

- If they have a different number of non-defaulted parameters
- If they have at least one parameter with different type

Function Overloading (2)

Indistinguishable parameter lists (invalid C++)

```
void foo(unsigned i);  
void foo(unsigned j); // parameter names do not matter  
void foo(unsigned i, unsigned j = 1);  
void foo(uint32_t i); // on x86_64
```

Valid example

```
void foo(unsigned i) { /* do something */ }  
void foo(float f) { /* do something */ }  
  
int main() {  
    foo(1u); // calls foo(unsigned)  
    foo(1.0f); // calls foo(float)  
}
```



Basic IO (1)

Facilities for printing to and reading from the console

- Use *stream objects* defined in `<iostream>` header
- `std::cout` is used for printing to console
- `std::cin` is used for reading from console

The left-shift operator can be used to write to `std::cout`

```
#include <iostream>
// -----
int main() {
    unsigned i = 42;
    std::cout << "The value of i is " << i << std::endl;
}
```

Basic IO (2)

The right-shift operator can be used to read from `std::cin`

```
#include <iostream>
// -----
int main() {
    std::cout << "Please enter a value: " << std::flush;
    unsigned v;
    std::cin >> v;
    std::cout << "You entered " << v << std::endl;
}
```

The `<iostream>` header is part of the C++ standard library

- Many more interesting and useful features
- More details later
- If you want to know more: Read the documentation!

Declarations and Definitions



Objects

One of the core concepts of C++ are objects.

- The main purpose of C++ programs is to interact with objects in order to achieve some goal
- Examples of objects are local and global variables
- Examples of concepts that are *not* objects are functions, references, and values

An object in C++ is a *region of storage* with certain properties:

- Size
- Alignment
- Storage duration (automatic, static, thread, dynamic)
- Lifetime
- Type
- Value
- Optionally, a name



Declaration Specifiers

Some declarations can also contain additional *specifiers*. The following lists shows a few common ones and their effect on storage duration and linkage.

Specifier	Global Func/Variable	Local Variable
<i>none</i>	static + external	automatic + none
<i>static</i>	static + internal	static + none
<i>extern</i>	static + external	static + external
<i>thread_local</i>	thread + ext/int	thread + none

inline Permit multiple definitions of the same function. Despite the name, has (almost) nothing to do with the inlining optimization. See the slides about the “One Definition Rule” for more information.



Namespaces

Larger projects may contain many names (functions, classes, etc.)

- Should be organized into logical units
- May incur name clashes
- C++ provides *namespaces* for this purpose

Namespace definitions

```
namespace identifier {  
    namespace-body  
}
```

Explanation

- *identifier* may be a previously unused identifier, or the name of a namespace
- *namespace-body* may be a sequence of declarations
- A name declared inside a namespace must be qualified when accessed from outside the namespace (`::` operator)



Declarations

C++ code that introduces a name that can then be referred to is called *declaration*. There are many different kinds of declarations:

- variable declarations: `int a;`
 - At global scope, use `extern int a;`
- function declarations: `void foo();`
- namespace declarations: `namespace A { }`
- using declarations: `using A::x;`
- class declarations: `struct S; class C;`
- template declarations: `template <typename T> void foo();`
- ...



Definitions

When a name is declared, it can be referenced by other code. However, most uses of a name also require the name to be *defined* in addition to be declared.

Formally, this is called *odr-use* and covers the following cases:

- The value of a variable declaration is read or written
- The address of a variable or function declaration is taken
- A function is called
- An object of a class declaration is used

Most declarations are also definitions, with some exceptions such as

- Any declaration with an **extern** specifier and no initializer
- Function declarations without function bodies
- Declaration of a class name (“forward declaration”)



One Definition Rule

One Definition Rule (ODR)

- At most one definition of a name is allowed *within one translation unit*
- Exactly one definition of every non-inline function or variable that is odr-used must appear *within the entire program*
- Exactly one definition of an inline-function must appear *within each translation unit* where it is odr-used
- Exactly one definition of a class must appear *within each translation unit* where the class is used and required to be complete

For subtleties and exceptions to these rules: See reference documentation



Header and Implementation Files (1)

When distributing code over several files it is usually split into *header* and *implementation* files

- Header and implementation files have the same name, but different suffixes (e.g. `.hpp` for headers, `.cpp` for implementation files)
- Header files contain only declarations that should be visible and usable in other parts of the program
- Implementation files contain definitions of the names declared in the corresponding header
- At least the header files should include some documentation

Header and Implementation Files (2)

Why do we separate headers and implementation files?

- A `.cpp` file usually uses “external” functions and variables that are defined in another translation unit
- To compile a translation unit to an object file, the compiler needs to know the *declarations* of the external names
- Often it does not need to know the *definitions*
- Interdependent `.cpp` files can be compiled independently and simultaneously
- When only the definition and not the declaration of a function changes, no other translation units have to be recompiled
- Conceptual separation between “API” (in header files) and “Implementation” (in implementation files)

Note: In some cases the compiler does need the full definition of a name

→ Have to put definitions in headers in that case.

Header Guards (1)

A file may transitively include the same header multiple times

- May lead to unintentional redefinitions
- It is infeasible (and often impossible) to avoid duplicating transitive includes entirely
- Instead: Header files themselves ensure that they are included at most once in a single translation unit

```
_____ path/A.hpp _____  
inline int foo() { return 1; }
```

```
_____ path/B.hpp _____  
#include "path/A.hpp"  
inline int bar() { return foo(); }
```

```
_____ main.cpp _____  
#include "path/A.hpp"  
#include "path/B.hpp" // ERROR: foo is defined twice
```


Header Guards (2)

Solution: Use header guards

```
_____ path/A.hpp _____  
  
// use any unique name, usually composed from the path  
#ifndef H_path_A  
#define H_path_A  
inline int foo() { return 1; }  
#endif
```

```
_____ path/B.hpp _____  
  
#ifndef H_path_B  
#define H_path_B  
#include "path/A.hpp"  
inline int bar() { return foo(); }  
#endif
```

Most compilers also support the non-standard `#pragma once` preprocessor directive. We recommend: Always use header guards.

Debugging C++ Programs with gdb

- Debugging by printing text is easy but most of the time not very useful
- Especially for multi-threaded programs a real debugger is essential
- For C++ the most used debugger is gdb (“GNU debugger”)
- It is free and open-source (GPLv2)
- For the best debugging experience a program should be compiled without optimizations (`-O0`) and with debug symbols (`-g`)
- The debug symbols help the debugger to map assembly instructions to the source code that generated them
- The documentation for gdb can be found here:
<https://sourceware.org/gdb/current/onlinedocs/gdb/>

Runtime Checks for Debugging

- Stepping through a buggy part of the program is often enough to identify the bug
- At least, it can help to narrow down the location of a bug
- Sometimes it is better to write code that checks if an invariant holds

The `assert` macro can be used for that:

- Defined in the `<cassert>` header
- Can be used to check a boolean expression
- Only enabled when the `NDEBUG` macro is *not* defined
- Automatically enabled in debug builds when using CMake

```
div.cpp
#include <cassert>
double div(double a, int b) {
    assert(b != 0);
    return a / b;
}
```

When this function is called with `b==0`, the program will crash with a useful error message.

Automatic Runtime Checks (“Sanitizers”)

- Modern compilers can automatically add several runtime checks, they are usually called *sanitizers*
- Most important ones:
 - Address Sanitizer (ASAN): Instruments memory access instructions to check for common bugs
 - Undefined-Behavior Sanitizer (UBSAN): Adds runtime checks to guard against many kinds of undefined behavior
- Because sanitizers add overhead, they are not enabled by default
- Should normally be used in conjunction with `-g` for debug builds
- Compiler option for gcc/clang: `-fsanitize=<sanitizer>`
 - `-fsanitize=address` for ASAN
 - `-fsanitize=undefined` for UBSAN
- Should be enabled by default in your debug builds, unless there is a very compelling reason against it

References, Arrays, and Pointers



Overview

Much of the power of C++ comes from the ability to define *compound types*

- Functions
- Classes
- References
- Arrays
- Pointers



Reference Declaration (1)

A reference declaration declares an alias to an already-existing object or function

- **Lvalue reference:** *type& declarator*
- **Rvalue reference:** *type&& declarator*
- Most of the time, *declarator* will simply be a name

References have some peculiarities

- There are no references to `void`
- There are no “null” references
- References are immutable (although the referenced object may be mutable)
- References are not objects, i.e. they do not necessarily occupy storage

Since references are not objects

- There are no references or pointers to references
- There are no arrays of references

Reference Declaration (2)

The `&` or `&&` tokens are part of the *declarator*, not the type

```
int i = 10;  
int& j = i, k = i; // j is reference to int, k is int
```

However, we may omit or insert whitespaces before and after the `&` or `&&` tokens

- Both `int& j = i;` and `int &j = i;` are valid C++
- By convention, we use the former notation (`int& j = i;`)
- To avoid confusion, statements should declare only one identifier at a time
- Very rarely, exceptions to this rule are necessary in the *init-statements* of `if` and `switch` statements as well as `for` loops

Lvalue References (1)

As an alias for existing objects

```
unsigned i = 10;
unsigned j = 42;
unsigned& r = i; // r is an alias for i

r = 21;          // modifies i to be 21
r = j;          // modifies i to be 42

i = 123;
j = r;          // modifies j to be 123
```

Lvalue References (2)

To implement pass-by-reference semantics for function calls

```
void foo(int& value) {  
    value += 42;  
}  
  
int main() {  
    int i = 10;  
    foo(i);    // i == 52  
    foo(i);    // i == 94  
}
```

Lvalue References (3)

To turn a function call into an lvalue expression

```
int global0 = 0;
int global1 = 0;

int& foo(unsigned which) {
    if (!which)
        return global0;
    else
        return global1;
}

int main() {
    foo(0) = 42; // global0 == 42
    foo(1) = 14; // global1 == 14
}
```

Rvalue References (1)

Can not (directly) bind to lvalues

```
int i = 10;  
int&& j = i; // ERROR: Cannot bind rvalue reference to lvalue  
int&& k = 42; // OK
```

Extend the lifetimes of temporary objects

```
int i = 10;  
int j = 32;  
  
int&& k = i + j; // k == 42  
k += 42;        // k == 84;
```

Rvalue References (2)

Allow overload resolution to distinguish between lvalues and rvalues

```
void foo(int& x);  
void foo(const int& x);  
void foo(int&& x);  
  
int& bar();  
int baz();  
  
int main() {  
    int i = 42;  
    const int j = 84;  
  
    foo(i);        // calls foo(int&)  
    foo(j);        // calls foo(const int&)  
    foo(123);      // calls foo(int&&)  
  
    foo(bar())    // calls foo(int&)  
    foo(baz())    // calls foo(int&&)  
}
```

References and CV-Qualifiers

References themselves cannot be cv-qualified

- However, the referenced type may be cv-qualified
- A reference to T can be initialized from a type that is less cv-qualified than T (e.g. `const int&` can be initialized from `int&`)

```
int i = 10;
const int& j = i;
int& k = j; // ERROR: binding reference of type int& to
            //          const int discards cv-qualifiers
j = 42;     // ERROR: assignment of read-only reference
```

Lvalue references to `const` also extend the lifetime of temporary objects

```
int i = 10;
int j = 32;
const int& k = i + j; // OK, but k is immutable
```



Dangling references

It is possible to write programs where the lifetime of a referenced object ends while references to it still exist.

- This can already happen when referencing objects with automatic storage duration
- Results in *dangling reference* and undefined behavior

Example

```
int& foo() {  
    int i = 42;  
    return i;    // ERROR: Returns dangling reference  
}
```



Array Declaration (1)

An array declaration declares an object of array type (also: C-style array)

- *type declarator*[*expression*]
- *expression* must be an expression which evaluates to an integral constant at **compile time**
- Again, due to weird parsing rules [*expression*] is part of the declarator
- *type*[*expression*] can be used as a type outside of declarators

For example: `T a[N];` for some type `T` and compile-time constant `N`

- `a` consists of `N` contiguously allocated elements of type `T`
- Elements are numbered `0`, ..., `N - 1`
- Elements can be accessed with the subscript operator `[]`, e.g. `a[0]`, ..., `a[N - 1]`
- Without an initializer, every element of `a` is uninitialized

Array Declaration (2)

Example

```
unsigned short a[10];  
  
for (unsigned i = 0; i < 10; ++i)  
    a[i] = i + 1;
```

Array objects are lvalues, but they cannot be assigned to

```
unsigned short a[10];  
unsigned short b[10];  
  
a = b; // ERROR: a is an array
```

Arrays cannot be returned from functions

```
int[] foo(); // ERROR
```

Array Declaration (3)

Elements of an array are allocated **contiguously** in memory

- Given `unsigned short a[10]`; containing the integers 1 through 10
- Assuming a 2-byte `unsigned short` type
- Assuming little-endian byte ordering

a[0]		a[1]		a[2]		a[3]		a[4]		a[5]		a[6]		a[7]		a[8]		a[9]	
01	00	02	00	03	00	04	00	05	00	06	00	07	00	08	00	09	00	0a	00
00		02		04		06		08		0a		0c		0e		10		12	
Address																			

Arrays are just dumb chunks of memory

- Out-of-bounds accesses are not automatically detected, and do not necessarily lead to a crash
- May lead to rather weird bugs
- Exist mainly due to compatibility requirements with C



size_t

C++ has a designated type for indexes and sizes: `std::size_t` from `<cstdint>`

- `size_t` is an unsigned integer type that is large enough to represent sizes and all possible array indexes on the target architecture
- The C++ language and standard library use `size_t` when handling indexes or sizes
- Generally, use `size_t` for array indexes and sizes
- Sometimes you can also use smaller integer types (e.g. `unsigned`) when working with small arrays



std::array

C-style arrays should be avoided whenever possible

- Use the `std::array` type defined in the `<array>` standard header instead
- Same semantics as a C-style array
- Optional bounds-checking and other useful features
- `std::array` is a *template type* with two template parameters (the element type and count)

Example

```
#include <array>

int main() {
    std::array<unsigned short, 10> a;
    for (size_t i = 0; i < a.size(); ++i)
        a[i] = i + 1; // no bounds checking
}
```



std::vector (1)

std::array is inflexible due to compile-time fixed size

- The std::vector type defined in the <vector> standard header provides dynamically-sized arrays
- Storage is automatically expanded and contracted as needed
- Elements are still stored contiguously in memory

Useful functions

- push_back – inserts an element at the end of the vector
- size – queries the current size
- clear – clears the contents
- resize – change the number of stored elements
- The subscript operator can be used with similar semantics as for C-style arrays

Familiarize yourself with the reference documentation on std::vector

std::vector (2)

Example

```
#include <iostream>
#include <vector>

int main() {
    std::vector<unsigned short> a;
    for (size_t i = 0; i < 10; ++i)
        a.push_back(i + 1);

    std::cout << a.size() << std::endl; // prints 10
    a.clear();
    std::cout << a.size() << std::endl; // prints 0

    a.resize(10); // a now contains 10 zeros
    std::cout << a.size() << std::endl; // prints 10

    for (unsigned i = 0; i < 10; ++i)
        a[i] = i + 1;
}
```



Range-For (1)

Execute a `for`-loop over a range

```
for (init-statement; range-declaration : range-expression)
    loop-statement
```

Explanation

- Executes *init-statement* once, then executes *loop-statement* once for each element in the range defined by *range-expression*
- *range-expression* may be an expression that represents a sequence (e.g. an array or an object for which `begin` and `end` functions are defined, such as `std::vector`)
- *range-declaration* should declare a named variable of the element type of the sequence, or a reference to that type
- *init-statement* may be omitted

Range-For (2)

Example

```
#include <iostream>
#include <vector>

int main() {
    std::vector<unsigned short> a;

    // no range-for, we need the index
    for (size_t i = 0; i < 10; ++i)
        a.push_back(i + 1);

    // range-for
    for (const unsigned short& e : a)
        std::cout << e << std::endl;
}
```


Storage of Objects

A “region of storage” has a physical equivalent

- Typically, objects reside in main memory, either on the stack or on the heap
- In simple programs, we almost exclusively deal with objects residing on the stack

Objects reside at some specific *location* in main memory

- This location can be identified by an *address* in main memory
- It is convenient to think of addresses as simple offsets from the beginning of the address space
- Pointers are a feature of C++ to obtain and interact with these addresses



Pointer Declaration (1)

A pointer declaration declares a variable of pointer type

- *type** *cv declarator*
- *declarator* may be any other declarator, except for a reference declarator
- *cv* specifies the *cv*-qualifiers of the pointer (not the pointed-to type), and may be omitted
- Analogous to reference declarations, the * token is part of the declarator, not the type

Notes

- A pointer to an object represents the address of the *first* byte in memory that is occupied by that object
- As opposed to references, pointers are themselves objects
- Consequently, pointers to pointers are allowed

Pointer Declaration (2)

Examples of valid pointer declarations

```
int* a;           // pointer to int
const int* b;    // pointer to const int
int* const c;    // const pointer to int
const int* const d; // const pointer to const int
```

Pointer-to-pointer declarations

```
int** e;           // pointer to pointer to int
const int* const* const f; // const pointer to const pointer
                        // to const int
```

Contraptions like the declaration of `f` are very rarely (if at all) necessary



The Address-Of Operator

In general, there exists no implicit conversion from a pointed-to type (e.g. `int`) to its pointer type (e.g. `int*`)

- In order to obtain a pointer to an object, the built-in unary address-of operator `&` has to be used
- Given an lvalue expression `a`, `&a` returns a pointer to the value of the expression
- The cv-qualification of `a` is retained

Example

```
int a = 10;
const int b = 42;
int* c = &a;           // OK: c points to a
const int* d = &b;    // OK: d points to b
int* e = &b;           // ERROR: invalid conversion from
                       // const int* to int*
```



The Indirection Operator

In general, there exists no implicit conversion from a pointer type (e.g. `int*`) to its pointed-to type (e.g. `int`)

- In order to access the pointed-to object, the built-in unary indirection operator `*` has to be used
- Given an expression `expr` of pointer type, `*expr` returns an lvalue reference to the pointed-to object
- The cv-qualifiers of the pointed-to type are retained
- Applying the indirection operator is also called *dereferencing* a pointer

Example

```
int a = 10;
int* c = &a;
int& d = *c; // reference to a
d = 123;     // a == 123
*c = 42;     // a == 42
```



Null Pointers

A pointer may not point to any object at all

- Indicated by the special value and corresponding literal `nullptr`
- Pointers of the same type which are both null pointers are considered equal
- It is undefined behavior to dereference a null pointer

Undefined behavior can lead to surprising results

foo.cpp

```
int foo(const int* ptr) {  
    int v = *ptr;  
  
    if (ptr == nullptr)  
        return 42;  
  
    return v;  
}
```

foo.o

```
foo(int*):  
    movl    (%rdi), %eax  
    ret
```



Array to Pointer Decay

Arrays and pointers have many similarities

- There is an implicit conversion from values of array type to values of pointer type
- The conversion constructs a pointer to the first element of an array
- The pointer type must be at least as cv-qualified as the array type

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr = array;

    std::cout << "The first element of array is ";
    std::cout << *ptr << std::endl;
}
```



The Subscript Operator

The subscript operator is defined on pointer types

- Treats the pointer as a pointer to the first element of an array
- Follows the same semantics as the subscript operator on array types

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr = array;

    std::cout << "The elements of array are";
    for (unsigned i = 0; i < 3; ++i)
        std::cout << " " << ptr[i];
    std::cout << std::endl;
}
```




Arithmetic on Pointers (1)

Some arithmetic operators are defined between pointers and integral types

- Treats the pointer as a pointer to some element of an array
- Adding i to a pointer moves the it i elements to the right
- Subtracting i from a pointer moves it i elements to the left
- In general, for a pointer p the expressions $p[i]$ and $*(p + i)$ are equivalent

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr = &array[1];

    std::cout << "The previous element is ";
    std::cout << *(ptr - 1) << std::endl;
    std::cout << "The next element is ";
    std::cout << *(ptr + 1) << std::endl;
}
```

Arithmetic on Pointers (2)

Special care has to be taken to only dereference valid pointers

- Especially important since it is valid to take the *past-the-end* pointer of an array or `std::vector`

Example

```
int main() {
    std::vector<int> v;
    v.resize(10);

    const int* firstPtr = &v[0]; // OK: valid pointer
    const int* lastPtr = &v[10]; // OK: past-the-end pointer

    int last1 = *lastPtr; // ERROR, might segfault
    int last2 = v[10]; // ERROR, might segfault
}
```

Arithmetic on Pointers (3)

Subtraction is defined between pointers

- Treats both pointers as pointers to some elements of an array
- Computes the number of elements between these two pointers

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};
    const int* ptr1 = &array[0];
    const int* ptr2 = &array[3]; // past-the-end pointer

    std::cout << "There are " << (ptr2 - ptr1) << " elements ";
    std::cout << "in array" << std::endl;
}
```



Comparisons on Pointers

The comparison operators are defined between pointers

- Interprets the addresses represented by the pointers as integers and compares them
- Only defined if the pointers point to elements of the same array

Example

```
#include <iostream>

int main() {
    int array[3] = {123, 456, 789};

    std::cout << "The elements of array are"
    for (const int* it = &array[0]; it < &array[3]; ++it)
        std::cout << " " << *it;
    std::cout << std::endl;
}
```



Void Pointers

Pointers to `void` are allowed

- A pointer to an object of any type can implicitly be converted to a pointer to `void`
- The void pointer must be at least as cv-qualified as the original pointer
- The pointer value (i.e. the address) is unchanged

Usage

- Used to pass objects of unknown type
- Extensively used in C interfaces (e.g. `malloc`, `qsort`, ...)
- Only few operations are defined on void pointers (mainly assignment)
- In order to use the pointed-to object, one must *cast* the void pointer to the required type



static_cast (1)

The `static_cast` conversion is used to cast between related types

```
static_cast< new_type > ( expression )
```

Explanation

- Converts the value of *expression* to a value of *new_type*
- *new_type* must be at least as cv-qualified as the type of *expression*
- Can be used to convert void pointers to pointers of another type
- Many more use cases (see reference documentation)

static_cast (2)

Void pointers

```
int i = 42;
void* vp = &i;
int* ip = static_cast<int*>(vp);
```

Other related types

```
int sum(int a, int b);
double sum(double a, double b);

int main() {
    int a = 42;
    double b = 3.14;

    double x = sum(a, b); // ERROR: ambiguous
    double y = sum(static_cast<double>(a), b); // OK
    int z = sum(a, static_cast<int>(b)); // OK
}
```



reinterpret_cast

The `reinterpret_cast` conversion is used to convert between unrelated types

```
reinterpret_cast < new_type > ( expression )
```

Explanation

- Interprets the underlying bit pattern of the value of *expression* as a value of *new_type*
- *new_type* must be at least as cv-qualified as the type of *expression*
- Usually does not generate any CPU instructions

Only a very restricted set of conversions is allowed

- A pointer to an object can be converted to a pointer to `std::byte`, `char` or `unsigned char`
- A pointer can be converted to an integral type (typically `uintptr_t`)
- Invalid conversions usually lead to **undefined behavior**



Strict Aliasing Rule (1)

It is **undefined behavior** to access an object using an expression of different type

- In particular, we are not allowed to access an object through a pointer to another type (pointer aliasing)
- Consequently, compilers typically assume that pointers to different types cannot have the same value
- There are very few exceptions to this rule

Strict Aliasing Rule (2)

```
foo.cpp
static int foo(int* x, double* y) {
    *x = 42;
    *y = 3.0;
    return *x;
}

int main() {
    int a = 0;
    double* y = reinterpret_cast<double*>(&a);
    return foo(&a, y);
}
```

Compiling this with `g++ -O1` will result in the following assembly

```
foo.o
main:
movl    $0, %eax
ret
```

Strict Aliasing Rule (3)

```
foo.cpp
static int foo(int* x, double* y) {
    *x = 42;
    *y = 3.0;
    return *x;
}

int main() {
    int a = 0;
    double* y = reinterpret_cast<double*>(&a);
    return foo(&a, y);
}
```

Compiling this with `g++ -O2` will result in the following assembly

```
foo.o
main:
movl    $42, %eax
ret
```



The sizeof Operator

The `sizeof` operator queries the size of the object representation of a type

```
sizeof( type )
```

Explanation

- The size of a type is given in bytes
- `sizeof(std::byte)`, `sizeof(char)`, and `sizeof(unsigned char)` return `1` by definition
- Depending on the computer architecture, there may be 8 or more bits in one byte (as defined by C++)



The `alignof` Operator

Queries the alignment requirements of a type

```
alignof( type )
```

Explanation

- Depending on the computer architecture, certain types must have addresses aligned to specific byte boundaries
- The `alignof` operator returns the number of bytes between successive addresses where an object of `type` can be allocated
- The alignment requirement of a type is always a power of two
- Important (e.g.) for SIMD instructions, where the programmer must explicitly ensure correct alignment
- Memory accesses with incorrect alignment leads to undefined behavior, e.g. SIGSEGV or SIGBUS (depending on architecture)

Usage Guidelines

When to use references

- Pass-by-reference function call semantics
- When it is guaranteed that the referenced object will always be valid
- When object that should be referenced is always the same

When to use pointers

- Only when absolutely necessary!
- When there may not be a pointed-to object (i.e. `nullptr`)
- When the pointer may change to a different object
- When pointer arithmetic is desired

In advanced C++ this gets more complicated:

- Decision is intricately related to *ownership* semantics
- We would actually like to avoid using raw pointers as much as possible
- There are standard library classes which encapsulate pointers

Troubleshooting

Pointers have a reputation of being highly error-prone

- It is very easy to obtain pointers that point to invalid locations
- Once such a pointer is dereferenced, a number of bad things can happen

Bad things that may happen

- The pointer pointed outside of the program's address space
 - The program will likely segfault immediately
- The pointer pointed outside of the intended memory region, but still inside the program's address space
 - The program might segfault immediately
 - ...or simply corrupt some memory, which might lead to problems later

With the right tools, debugging is not as daunting as it may seem

The Infamous Segfault (1)

Every C++ programmer will encounter a segfault eventually

- Raised by hardware in response to a memory access violation
- In most cases caused by invalid pointers or memory corruption

Obvious example

```
foo.cpp  
  
int main() {  
    int* a;  
    return *a; // ERROR: Dereferencing an uninitialized pointer  
}
```

Executing this program might result in the following

```
└─$  
$ ./foo  
[1] 5128 segmentation fault (core dumped) ./foo
```


The Infamous Segfault (2)

Sometimes, the root cause may be (much) more difficult to determine

```
bar.cpp  
  
int main() {  
    long* ptr;  
    long array[3] = {123, 456, 789};  
    ptr = &array[0];  
    array[3] = 987; // ERROR: off-by-one access  
  
    return *ptr;  
}
```

When compiled with `g++ -fno-stack-protector`, this will also segfault

- The off-by-one access `array[3] = 987` actually changes the value of `ptr`
- Dereferencing this pointer in the return statement will result in a segfault
- The `-fno-stack-protector` option is required, because `g++` will by default emit extra code to prevent such buffer overflows

The Infamous Segfault (3)

Use the address sanitizer!



```
$ g++ -g -fno-stack-protector -obar bar.cpp
$ ./bar
[1] 4199 segmentation fault (core dumped) ./bar
$ g++ -g -fno-stack-protector -fsanitize=address -obar bar.cpp
$ ./bar
=====
==4229==ERROR: AddressSanitizer: stack-buffer-overflow on address [...]
WRITE of size 8 at 0x7fff536479d8 thread T0
#0 0x5617976d529f in main (/tmp/bar+0x129f)
#1 0x7f6a0fcc3022 in __libc_start_main (/usr/lib/libc.so.6+0x27022)
#2 0x5617976d50ad in _start (/tmp/bar+0x10ad)

Address 0x7fff536479d8 is located in stack of thread T0 at offset 56 in
↳ frame
#0 0x5617976d5188 in main (/tmp/bar+0x1188)

This frame has 1 object(s):
[32, 56) 'array' (line 3) <== Memory access at offset 56 overflows
↳ this variable

[...]
==4229==ABORTING
```

Classes



Classes

In C++ classes are the main kind of user-defined type.

Informal specification of a class definition:

```
class-keyword name {  
    member-specification  
};
```

- *class-keyword* is either **struct** or **class** (only difference: visibility of members)
- *name* can be any valid identifier (like for variables, functions, etc.)
- *member-specification* is a list of declarations, mainly variables (“data members”), functions (“member functions”), and types (“nested types”)
- The trailing semicolon is mandatory!



Data Members

- Declarations of data members are variable declarations
- `extern` is not allowed
- Declarations without `static` are called *non-static* data members, otherwise they are *static* data members
- `thread_local` is only allowed for static data members
- Declaration must have a *complete type* (see later slide)
- Name of the declaration must differ from the class name and must be unique within the class
- Non-static data members can have a *default value*

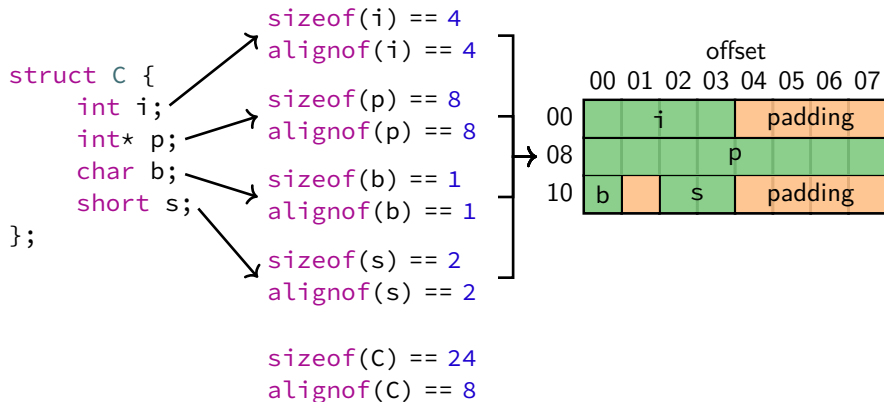
```
struct Foo {  
    // non-static data members:  
    int a = 123;  
    float& b;  
    const char c;  
    // static data members:  
    static int s;  
    thread_local static int t;  
};
```

Memory Layout of Data Members (Standard-Layout)



- Every type has a size and an alignment requirement
- To be compatible between different compilers and programming languages (mainly C), the memory layout of objects of class type is fixed, if all non-static data members have the same access control and the class is a *standard-layout class*
- Non-static data members appear in memory by the order of their declarations
- Size and alignment of each data-member is accounted for → leads to “gaps” in the object, called *padding bytes*
- Alignment of a class type is equal to the largest alignment of all non-static data members
- Size of a class type is at least the sum of all sizes of all non-static data members and at least 1
- static data members are stored separately

Size, Alignment and Padding



Reordering the member variables in the order p, i, s, b would lead to `sizeof(C) == 16!`

In general: Order member variables by decreasing alignment to get the fewest padding bytes.



Member Functions

- Declarations of member functions are like regular function declarations
- Just like for data members, there are non-static and static (with the `static` specifier) member functions
- Non-static member functions can be *const-qualified* (with `const`) or *ref-qualified* (with `const&`, `&`, or `&&`)
- Non-static member functions can be `virtual`
- There are some member functions with special functions:
 - Constructor and destructor
 - Overloaded operators

```
struct Foo {  
    void foo(); // non-static member function  
    void cfoo() const; // const-qualified non-static member function  
    void rfoo() &; // ref-qualified non-static member function  
    static void bar(); // static member function  
    Foo(); // Constructor  
    ~Foo(); // Destructor  
    bool operator==(const Foo& f); // Overloaded operator ==  
};
```




Accessing Members

Given the following code:

```
struct C {  
    int i;  
    static int si;  
};  
C o; // o is variable of type C  
C* p = &o; // p is pointer to o
```

the members of the object can be accessed as follows:

- non-static and static member variables and functions can be accessed with the *member-of* operator: `o.i`, `o.si`
- As a shorthand, instead of writing `(*p).i`, it is possible to write `p->i`
- Static member variables and functions can also be accessed with the *scope resolution* operator: `C::si`

Other User-Defined Types



Unions

- In addition to regular classes declared with `class` or `struct`, there is another special class type declared with `union`
- In a union only one member may be “active”, all members use the same storage
- Size of the union is equal to size of largest member
- Alignment of the union is equal to largest alignment among members
- Strict aliasing rule still applies with unions!
- Most of the time there are better alternatives to unions, e.g. `std::array<std::byte, N>` or `std::variant`

```
union Foo {  
    int a;  
    double b;  
};  
sizeof(Foo) == 8;  
alignof(Foo) == 8;
```

```
Foo f; // No member is active  
f.a = 1; // a is active  
std::cout << f.b; // Undefined behavior!  
f.b = 12.34; // Now, b is active  
std::cout << f.b; // OK
```



Enums

- C++ also has user-defined enumeration types
- Typically used like integral types with a restricted range of values
- Also used to be able to use descriptive names instead of “magic” integer values
- Syntax: *enum-key name { enum-list };*
- *enum-key* can be **enum**, **enum class**, or **enum struct**
- *enum-list* consists of comma-separated entries with the following syntax:
name [= value]
- When *value* is not specified, it is automatically chosen starting from 0

```
enum Color {  
    Red, // Red == 0  
    Blue, // Blue == 1  
    Green, // Green == 2  
    White = 10,  
    Black, // Black == 11  
    Transparent = White // Transparent == 10  
};
```

Using Enum Values

- Names from the enum list can be accessed with the scope resolution operator
- When `enum` is used as keyword, names are also introduced in the enclosing namespace
- Enums declared with `enum` can be converted implicitly to `int`
- Enums can be converted to integers and vice versa with `static_cast`
- `enum class` and `enum struct` are equivalent
- Guideline: Use `enum class` unless you have a good reason not to

```
Color::Red; // Access with scope resolution operator
Blue; // Access from enclosing namespace
int i = Color::Green; // i == 2, implicit conversion
int j = static_cast<int>(Color::White); // j == 10
Color c = static_cast<Color>(11); // c == Color::Black
```

Further Topics

Further Topics

C++ has many more features and components:

- Constructor, Destructors, Assignment
- Dynamic memory management, Ownership
- Class inheritance, dynamic dispatch
- Templates, Concepts
- Standard library
- Atomics, mutexes, threads
- ...

→ cppreference.com should always be your first place to look

In Summer we offer a practical course on C++!