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
- Lifetime
- Type
- Value
- Optionally, a name

Storage Duration (1)

Every object has one of the following *storage durations*:

automatic:

- Objects with automatic storage duration are allocated at the beginning of the enclosing scope and deallocated automatically (i.e., it is not necessary to write code for this) at its end
- Local variables have automatic storage duration

static:

- Objects with static storage duration are allocated when the program begins (usually even before `main()` is executed!)
- They are deallocated automatically when the program ends
- All global variables have static storage duration
- The order of construction of different variables is not guaranteed $\rightarrow$ can easily lead to unexpected bugs. (See also: Static Initialization Order Fiasco).

Storage Duration (2)

thread:

- Objects with thread storage duration are allocated when a thread starts and deallocated automatically when it ends
- In contrast to objects with static storage duration, each thread gets its own copy of objects with thread storage duration

dynamic:

- Objects with dynamic storage duration are allocated and deallocated by using dynamic memory management
- Note: Deallocation must be done manually!

```
int foo = 1; // static storage duration
static int bar = 2; // static storage duration
thread_local int baz = 3; // thread storage duration
void f() {
    int x = 4; // automatic storage duration
    static int y = 5; // static storage duration
}
```

Lifetime

In addition to their storage duration objects also have a *lifetime* which is closely related. References also have a lifetime.

- The lifetime of an object or reference starts when it was fully *initialized*
- The lifetime of an object ends when its destructor is called (for objects of class types) or when its storage is deallocated or reused (for all other types)
- The lifetime of an object never exceeds its storage duration.
- The lifetime of a reference ends as if it were a "scalar" object (e.g. an `int` variable)

Generally, using an object outside of its lifetime leads to undefined behavior.

Lifetime issues are the main source of memory bugs!

- A C++ compiler can only warn about very basic lifetime errors
- If the compiler warns, always fix your code so that the warning disappears

Linkage

Most declarations have a (conceptual) property called *linkage*. This property determines how the name of the declaration will be visible in the current and in other translation units. There are three types of linkage:

no linkage:

- Names can only be referenced from the scope they are in
- Local variables

internal linkage:

- Names can only be referenced from the same translation unit
- Global functions and variables declared with `static`
- Global variables that are not declared with `extern`
- All declarations in namespaces without name ("anonymous namespaces")

external linkage:

- Names can be referenced from other translation units
- Global functions (without `static`)
- Global variables with `extern`

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. We will see some more specifiers in future lectures.

Specifier	Global Func/Variable	Local Variable
none	static + external	automatic + none
`static`	static + internal	static + none
`extern`	static + external	static + external
`thread_local`	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 (1)

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)

Namespaces (2)

Qualified name lookup

```
namespace A {
void foo() { /* do something */ }
void bar() {
    foo();  // refers to A::foo
}
}
namespace B {
void foo() { /* do something */ }
}
int main() {
    A::foo();  // qualified name lookup
    B::foo();  // qualified name lookup

    foo();     // ERROR: foo was not declared in this scope
}
```

Namespaces (3)

Namespaces may be nested

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

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

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

Namespaces (4)

Code can become rather confusing due to large number of braces

- Use visual separators (comments) at sensible points
- (Optionally) add comments to closing namespace braces

```
//------------------------------------
namespace A::B {
//------------------------------------
void foo() {
    // do something
}
//------------------------------------
void bar() {
    // do something else
}
//------------------------------------
}  // namespace A::B
//------------------------------------
```

Namespaces (5)

- Always using fully qualified names makes code easier to read
- Sometimes it is obvious from which namespace the names come from in which case one prefers to use unqalified names
- For this `using` and `using namespace` can be used
- `using namespace X` imports *all* names from namespace `X`
- `using X::a` only imports the name `a` from `X` into the current namespace
- Should not be used in header files to not influence other implementation files

```
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
}
```

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: `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 (1)

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

One Definition Rule (2)

a.cpp

```
int i = 5;  // OK: declares and defines i
int i = 6;  // ERROR: redefinition of i

extern int j; // OK: declares j
int j = 7;    // OK: (re-)declares and defines j
```

Separate declaration and definition is required to break circular dependencies

b.cpp

```
void bar();  // declares bar
void foo() { // declares and defines foo
    bar();
}
void bar() { // (re-)declares and defines bar
    foo();
}
```

One Definition Rule (3)

a.cpp

```
int foo() {
    return 1;
}
```

b.cpp

```
int foo() {
    return 2;
}
```

Trying to link a program consisting of a.cpp and b.cpp will fail

```
$ g++ -c -o a.o a.cpp
$ g++ -c -o b.o b.cpp
$ g++ a.o b.o
/usr/bin/ld: b.o: in function `foo()':
b.cpp:(.text+0x0): multiple definition of `foo()'; a.o:a.cpp:(.text+0x0): first
↪  defined here
collect2: error: ld returned 1 exit status
```

One Definition Rule (4)

What about helper functions/variables local to translation units? → Internal linkage!

- Option A: Use `static` (only works for variables and functions)

a.cpp

```
static int foo = 1;
static int bar() {
    return foo;
}
```

- Option B: Use *anonymous namespaces*

b.cpp

```
namespace {
//---------------------------------
int foo = 1;
int bar() {
    return foo;
}
//---------------------------------
}
```

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
$\rightarrow$ 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.

Example: Header and Implementation Files (1)

The example CMake project from last lecture shows how header and implementation files are used. These are the header files:

sayhello.hpp

```
#ifndef H_exampleproject_sayhello
#define H_exampleproject_sayhello
#include <string_view>
/// Print a greeting for `name`
void sayhello(std::string_view name);
#endif
```

saybye.hpp

```
#ifndef H_exampleproject_saybye
#define H_exampleproject_saybye
#include <string_view>
/// Say bye to `name`
void saybye(std::string_view name);
#endif
```

Example: Header and Implementation Files (2)

The two header files have the following associated implementation files:

sayhello.cpp

```cpp
#include "sayhello.hpp"
#include <iostream>

/// Print a greeting for `name`
void sayhello(std::string_view name) {
    std::cout << "Hello " << name << '!' << std::endl;
}
```

saybye.cpp

```cpp
#include "saybye.hpp"
#include <iostream>

/// Say bye to `name`
void saybye(std::string_view name) {
    std::cout << "Bye " << name << '!' << std::endl;
}
```

Example: Header and Implementation Files (3)

The "main" file, in the example `print_greetings.cpp` only includes the headers:

print_greetings.cpp

```
#include <iostream>
#include "sayhello.hpp"
#include "saybye.hpp"

int main(int argc, const char** argv) {
    if (argc != 2) {
        std::cerr << "Please write: ./print_greetings name"
        ↪  << std::endl;
        return 1;
    }
    sayhello(argv[1]);
    saybye(argv[1]);
    return 0;
}
```

Compiling C++ files

Hello World 2.0

In C++ the code is usually separated into *header files* (`.h`/`.hpp`) and *implementation files* (`.cpp`/`.cc`):

sayhello.hpp

```
#include <string_view>
void sayhello(std::string_view name);
```

sayhello.cpp

```
#include "sayhello.hpp"
#include <iostream>
void sayhello(std::string_view name) {
    std::cout << "Hello " << name << '!' << std::endl;
}
```

Other code that wants to use this function only has to include `sayhello.hpp`.

Compiler

Reminder: Internally, the compiler is divided into Preprocessor, Compiler, and Linker.

Preprocessor:

- Takes an input file of (almost) any programming language
- Handles all *preprocessor directives* (i.e., all lines starting with #) and *macros*
- Outputs the file without any preprocessor directives or macros

Compiler:

- Takes a preprocessed C++ (or C) file, called *translation unit*
- Generates and optimizes the machine code
- Outputs an *object file*

Linker:

- Takes multiple object files
- Can also take references to other libraries
- Finds the address of all symbols (e.g., functions, global variables)
- Outputs an executable file or a shared library

Preprocessor (1)

Preprocessor directive `#include`: Copies (!) the contents of a file into the current file.

Syntax:

- `#include "path"` where *path* is a relative path to the header file
- `#include <path>` like the first version but only *system directories* are searched for the *path*

In C++ usually only header files are included, never `.cpp` files!

Preprocessor (2)

Preprocessor directive `#define`: Defines a macro.

Syntax:

- `#define FOO`: Defines the macro `FOO` with no content
- `#define BAR 1`: Defines the macro `BAR` as 1

Before the compiler sees the file, all occurrences of `FOO` will be removed, `BAR` will be replaced with 1.

Note: Don't use this as "constant variables", use `constexpr` global variables instead!

Preprocessor (3)

Preprocessor directives `#ifdef`/`#ifndef`/`#else`/`#endif`: Removes all code up to the next `#else`/`#endif` if a macro is set (`#ifdef`) or not set (`#ifndef`)

Example:

```
#ifdef FOO
...
#endif
```

Mainly used for *header guards*.

Compiler

- Every *translation unit* (usually a `.cpp` file) results in exactly one object file (usually `.o`)
- References to external symbols (e.g., functions that are defined in another `.cpp`) are *not* resolved

mul.cpp

```
int add(int a, int b);
int mul(int a, int b) {
    if (a > 0) { return add(a, mul(a - 1, b)); }
    else { return 0; }
}
```

Assembly generated by the compiler:

```
_Z3mulii:
    testl   %edi, %edi
    jle     .L2
    pushq   %rbx
    movl    %edi, %ebx
    leal    -1(%rdi), %edi
    call    _Z3mulii
    movl    %ebx, %edi
    popq    %rbx
    movl    %eax, %esi
    jmp     _Z3addii@PLT
.L2:
    xorl    %eax, %eax
    ret
```

You can try this out yourself at `https://compiler.db.in.tum.de`

Linker

- The linker usually does not have to know about any programming language
- Still, some problems with your C++ code will only be found by the linker and not by the compiler (e.g., ODR violations)
- Most common error are missing symbols, happens either because you forgot to define a function or global variable, or forgot to add a library
- Popular linkers are: GNU `ld`, GNU `gold`, `lld` (by the LLVM project)

Compiler Flags (2)

- Preprocessor and linker are usually executed by the compiler
- There are additional compiler flags that can influence the preprocessor or the linker

Advanced flags:

`-E`	Run only preprocessor (outputs C++ file without preprocessor directives)
`-c`	Run only preprocessor and compiler (outputs object file)
`-S`	Run only preprocessor and compiler (outputs assembly as text)
`-g`	Add *debug symbols* to the generated binary
`-DFOO`	Defines the macro `FOO`
`-DFOO=42`	Defines the macro `FOO` with value 42
`-l<lib>`	Link library `<lib>` into executable
`-I<path>`	Also search `<path>` for `#included` files
`-L<path>`	Also search `<path>` for libraries specified with `-l`

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/`

gdb commands (1)

To start debugging run the command `gdb myprogram`. This starts a command-line interface wich expects one of the following commands:

`help`	Show general help or help about a command.
`run`	Start the debugged program.
`break`	Set a breakpoint. When the breakpoint is reached, the debugger stops the program and accepts new commands.
`delete`	Remove a breakpoint.
`continue`	Continue running the program after it stopped at a breakpoint or by pressing Ctrl + C .
`next`	Continue running the program until the next source line of the current function.
`step`	Continue running the program until the source line changes.
`nexti`	Continue running the program until the next instruction of the current function.
`stepi`	Execute the next instrution.
`print`	Print the value of a variable, expression or CPU register.

gdb commands (2)

`frame`	Show the currently selected *stack frame*, i.e. the current stack with its local variables. Usually includes the function name and the current source line. Can also be used to switch to another frame.
`backtrace`	Show all stack frames.
`up`	Select the frame from the next higher function.
`down`	Select the frame from the next lower function.
`watch`	Set a watchpoint. When the memory address that is watched is read or written, the debugger stops.
`thread`	Show the currently selected thread in a multi-threaded program. Can also be used to switch to another thread.

Most commands also have a short version, e.g., `r` for `run`, `c` for `continue`, etc.

Runtime Checks for Debugging

- Stepping though 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

UBSAN Example

foo.cpp

```cpp
#include <iostream>
int main() {
    int a; int b;
    std::cin >> a >> b;
    int c = a * b;
    std::cout << c << std::endl;
    return 0;
}
```

```
$ g++ -std=c++20 -g -fsanitize=undefined foo.cpp -o foo
$ ./foo
123456
789123
foo.cpp:7:9: runtime error: signed integer overflow: 123456 *
↪  789123 cannot be represented in type 'int'
-1362278720
```