Effective C++ Programming

NIE-EPC (v. 2021):

INTRODUCTION, OBSERVABLE BEHAVIOR, AS-IF RULE, OPTIMIZATIONS, TRANSLATION UNITS, ABI

Programming languages

- Programming — writing some functionality with a special syntax (*source code*).
- “Normal” programs — execution of the functionality by a CPU.
- CPUs understand only a single “language” — machine code.
- *Implication:* source code must be translated into machine code.
  - The role of programming languages implementations (*compilers, interpreters, libraries, linkers,…*).
- *Example:*

```
// C++ function source code
int absmax(int a, int b)
{
  int abs_a = std::abs(a);
  int abs_b = std::abs(b);
  return std::max(abs_a, abs_b);
}
```

```
// function machine code in hex...
// ... generated by GCC on x86_64:
89 fa 89 f0 c1
fa 1f 31 d7 29
d7 99 31 d0 29
d0 39 c7 0f 4d
c7 c3
```

Abstraction

- How do programming languages differ?
- *Among others (different syntax, target application domain,…),* they provide different sets of abstraction mechanisms.
- **Abstraction**:
  - A programming language concept that makes expressing the required functionality easier.
  - *Examples:* variables, function parameters, function return values, loop constructs, structures, classes, inheritance, polymorphism, templates, reflection,…
  - None of these concepts exist at a machine code level.

```cpp
// C++ function source code
int absmax(int a, int b)
{
  int abs_a = std::abs(a);
  int abs_b = std::abs(b);
  return std::max(abs_a, abs_b);
}
```

```
89 fa 89 f0 c1 fa
1f 31 d7 29 d7 99
31 d0 29 d0 39 c7
0f 4d c7 c3
// machine instruction ONLY for...
// ... register operations...
// ... plus final ret
```

Abstraction levels

- Classification of programming languages according to abstraction mechanisms they provide:
- **Low-level programming languages**:
  - Low-level = simple abstraction mechanisms (*variables*, *function parameters*,…).
- **High-level programming languages**:
  - High-level = complex and more powerful abstraction mechanisms (*classes*, *polymorphism*, *templates*,…).
  - Allow us to express some functionality more easily…
  - …at a price of loosing a control over a system.
- **Low-level programming languages *(cont.)***:
  - More effort to express some functionality…
  - …but provide better system control.
- *Generally:* the higher level of abstraction, the less system control, and vice versa.

Exemplary languages

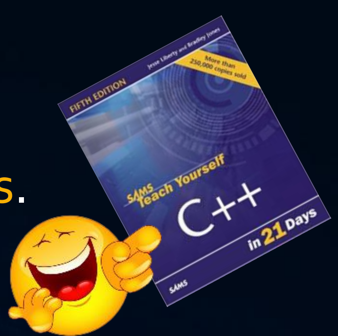
- **Assembler**
  - the lowest-level programming language,
  - total system control — 1-to-1 correspondence between assembler instructions and machine code instructions,
  - extreme effort to write some functionality,
  - not portable.
- **C**
  - portable,
  - large to moderate effort to write some functionality,
  - relatively high level of system control due to low-level abstraction mechanisms (*pointers*, *atomics*, *memory model control*).
- **Python**
  - easy to learn, easy to write some functionality with,
  - almost no control of what happens at a machine code level (no pointers, no atomics,…).

C++

- **C++ unique feature**:
  - widely adopted language that…
  - … has high-level abstraction mechanisms…
  - … and provide high-level of system control at the same time.
- **High level of abstraction** — a relatively easy way how to put together some functionality, if all necessary building blocks are available.
  - *Example:* Writing a program that works with strings is easy by using the `std::string` class (no need to understand what happens at the machine code level).
- **High level of system control** — development of building blocks with maximum performance and efficiency.
  - *Example (cont.):* implementing `std::string` efficiently requires high level of system control (*pointers, stack/heap allocations, alignment, cache system, memory representation bits manipulation,…*).

C++ *(cont.)*

- **Control = opposite of safety**:
  - More control ⇒ more we can do with the system ⇒ more bad things we can do with the system.
  - *"Writing in C or C++ is like running a chain saw with all the safety guards removed." [Bob Gray]*
- **Control over system = control over performance ≠ performance**:
  - *"C++ doesn't give you performance, it gives you control over performance." [Chandler Carruth (?)]*
  - Performance is not provided automatically by C++; it's easy to write an inefficient and slow program.
  - Doing the opposite requires a high level of knowledge and experience.
- **Mastering C++ is very hard!**
  - The latest C++ Standard has over 1850 pages.



C++ standards

- Definition of C++ — C++ standards.
- Evolution of C++ — multiple standards:
  - “Old C++”: C++98, C++03
  - “Modern C++”: C++11, C++14, C++17, C++20 (latest), C++23 (upcoming)
- New standards are mostly backwards-compatible.
- “Release” versions are paid ISO/IEC documents.
- Working drafts are available, e.g., here: [*hyperlink*]
  - Prior to publication, drafts were/are referred to as *C++1x, … ,C++2b.*
- Standards are (try to be) comprehensive and exact.
  - *Implication:* they are hard to read (*like a law code*).
- Better for learning: books, tutorials, lectures, language references, Stack Overflow,…
  - These sources may contain imprecise, misleading, or even incorrect information.
  - When you are in trouble or have doubts, always check with the C++ standard.

C++ standards *(cont.)*

C++ Standards have 2 main parts:

1. **C++ language**
   - Defines core language.
   - Exemplary entities: *types, variables, functions, classes, templates,…*
2. **C++ standard library**
   - Defines some useful building blocks for different functionality domains.
   - Exemplary entities: *algorithms, containers, iterators, smart pointers, threading, I/O streams,…*
   - No implementation details — C++ standards define only API (*application programming interface*) for library entities and optionally some requirements for their implementation.
   - *Example:* `std::sort` — API + the requirement for *O(n log n)* time complexity on average ⇒ can be implemented with *quicksort*, *heapsort*,… but not with *insertion sort*, *bubblesort* etc.

How do C++ standards define C++?

- C++ Standards define how source code should behave when it is translated and executed on an abstract machine.
- **Abstract machine** — an abstract (*imaginary, virtual*) computer system with a relatively small set of capabilities:
  - abstract CPU — arithmetic operations, bitwise operations,…;
  - memory — static allocations, dynamic allocations, addressing,…;
  - files — reading/writing data;
  - file system (*since C++17*);
  - standard input/output/error streams;
  - threading and memory model (*since C++11*),
  - some other abstract operation system mechanisms (program arguments, program exit code, signals,…).
- No screen, no keyboard, no mouse, no camera, no GPU, no networking, no heap, no stack, no instruction set, no caches, no cores,…

Observable behavior

- Prescribed behavior for some code = effects of this code observable from outside of this code, that is, from the perspective of its exterior/surroundings.
- The term “observable behavior” is typically used.
- *Example:*

```
void f()
{
  std::cout << "some string";
}
```

- Observable behavior of function `f`:
  1. writing the string “some string” into the program output stream,
  2. implicit return = returning the control flow to the function caller.
- This behavior may be observed from outside of the function:
  1. by reading the standard output stream on the abstract machine,
  2. by the function caller.

Observable behavior *(cont.)*

- *Another example:*

```
void g() {
  int a = 1, b = 2, c;  // declaration (+initialization) of local variables
  c = a + b;            // addition and assignment
}
```

- Observable behavior of function g:
  - implicit `return` = returning the control flow to the function caller;
  - nothing of the explicitly written code inside the function body contributes to its observable behavior (function-local non-static variables cannot be accessed from outside of this function).
- *Yet another example:*

```
int h() {
  int a = 1, b = 2, c;
  c = a + b;
  return c;
}
```

- Observable behavior of function h:
  - Returning the value 3 to the function caller.
  - Nothing else inside its definition contributes to its observable behavior.

Observable behavior *(cont.)*

- The boundary between observable and non-observable behavior is not always clear.
- **Observable behavior:**
  - reading/writing standard input/output/error streams;
  - writing data to files;
  - passing return values from functions to their callers;
  - modifying data in memory, if this memory may be accessed from outside;
  - updates of objects of *volatile* and *atomic* data types;
  - program termination with some exit code;
  - etc.
- **Hard case example:**
  - dynamic memory allocations.

Observable behavior *(cont.)*

- *Example:*

```
int main() {
  int* ptr = new int(1);
  delete ptr;
}
```

- Observable behavior of function `main`:
  - Implicit "`return 0`" at the end (*since C++11*).
- **Nothing else?** — The allocated memory cannot be accessed from outside of the function.
- *Controversy:* memory allocation may be observed from outside of the code (e.g., by measuring program memory consumption, by memory debuggers like *Valgrind* or *Heaptrack*, etc.).
- In the past, dynamic memory allocations were mostly considered as a part of the observable behavior.
- Now, "mostly" not:
  - *Experiments:* GCC since version 10, Clang since version 3.2.

Observable behavior *(cont.)*

- *…example from the previous slide:*

```
int main() {
   int* ptr = new int(1);
   delete ptr;
}
```

- *Another example:*

```
int* f() {
   int* ptr = new int(1);
   return ptr;
}
```

- Observable behavior of function `f`:
  - Returning a pointer to dynamically allocated object of type `int` with value 1.
- The effect of both:
  - dynamic memory allocation and
  - initialization of the allocated object to 1
- is observable from outside of the function.

Observable behavior *(cont.)*

- *Yet another example:*

```
int* f() {
  int* ptr = new int(1);
  return ptr;
}
int main() {
  int* ptr = f();
  delete ptr;
}
```

- Observable behavior of function `f` is the very same as on the previous slide:
  - Returning a pointer to dynamically allocated object of type `int` with value 1.
- Observable behavior of function `main`:
  - Implicit "`return 0`" only, provided that allocations themselves are not considered to contribute to the observable behavior (*as with latest GCC and Clang*).
  - Calling `f` itself cannot be observed from outside of `main`.

C++ implementation

- C++ standard — defines observable behavior for some code executed on the abstract machine.
- How does C++ work on a real computer system?
- There, we need a **C++ implementation**:
  - A tool that for some source code yields on a particular computer system the same observable behavior as is prescribed to it by the C++ standard.
- Typically, a collection of tools rather than a single tool only.
  - CPUs do not understand C++ source code $\Rightarrow$ it must be translated into machine code.
  - Such translation is the main purpose of a **C++ compiler**.
  - Mainstream C++ compiler vendors:
    GNU/GCC, LLVM/Clang, Microsoft/MSVC, Intel, IBM, PGI, Cray,…

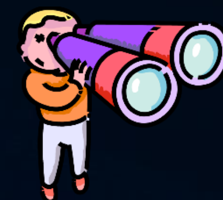
C++ implementation *(cont.)*

- C++ implementation = C++ compiler only? Not at all…
- **Preprocessor:**
  - Processes preprocessor directives and symbols (`#include`, `#define`, *macros, conditional compilation,…*).
- **Linker:**
  - Links different parts of machine code together.
  - Generates the final binary executable file (with a particular structure given by the system, such as ELF on Linux).
- **Implementation of the C++ standard library:**
  - Implements the C++ Standard library API defined by C++ standards with respect to their requirements.
  - May or may not be bound to a compiler.
  - Mainstream implementations: *libstdc++ (GNU), libc++ (LLVM), Microsoft STL* (*note:* their source code is available on *GitHub*).
- **And possibly others…**

As-if rule

- One of the most fundamental and essential C++ concept.
- C++ standard prescribes observable behavior for some source code when it is executed on the abstract machine.
- For the same source code, a C++ implementation generates machine code executable on a particular computer system.
- Conforming implementation: the observable behavior of the generated machine code **must be the same** as the observable behavior prescribed to the source code by the C++ standard.
- The implementation must guarantee that the machine code will **observably behave as if** the source code was executed on the abstract machine $\Rightarrow$ know as the "***as-if rule***".

```
// abstract machine
int absmax(int a, int b)
{
  int abs_a = std::abs(a);
  int abs_b = std::abs(b);
  return std::max(abs_a, abs_b);
}
```



```
// GCC/x86_64/Linux
89 fa 89 f0 c1 fa
1f 31 d7 29 d7 99
31 d0 29 d0 39 c7
0f 4d c7 c3
```

Optimizations

- The as-if rule:
  - The standard tells the implementation what behavior it should provide for some source code.
  - It does not tell anything about how this behavior should be provided.
- How implementations work in this regard? Generally, they are two options:

1. **Disabled optimizations**
   - *Semantics:* a user does not care about performance and efficiency.
   - *Typical situation:* debugging (debug builds).
   - *Example:* `g++` or `clang++` with `-O0` command line option.
2. **Enabled optimizations**
   - *Semantics:* a user needs performance and efficiency.
   - *Typical situations:* release (production) builds.
   - *Example:* `g++` or `clang++` with `-O2` command line option.

Optimizations *(cont.)*

- Effects:

1. Disabled optimizations:
   - a compiler generates machine code for everything that is in the source code.
2. Enabled optimizations:
   - a compiler generates optimal machine code for observable behavior only.

- *Example (Clang/Linux/x86_64):*

```
void g()
{
  int a = 1;
  int b = 2;
  int c;
  c = a + b;
  // implicit return
}
```

```
// machine code
// compiled with
// clang++ -O0:
55 48 89 e5 c7 45
fc 01 00 00 00 c7
45 f8 02 00 00 00
8b 45 fc 03 45 f8
89 45 f4 5d c3
```

```
// machine code
// compiled with
// clang++ -O2:
c3
```

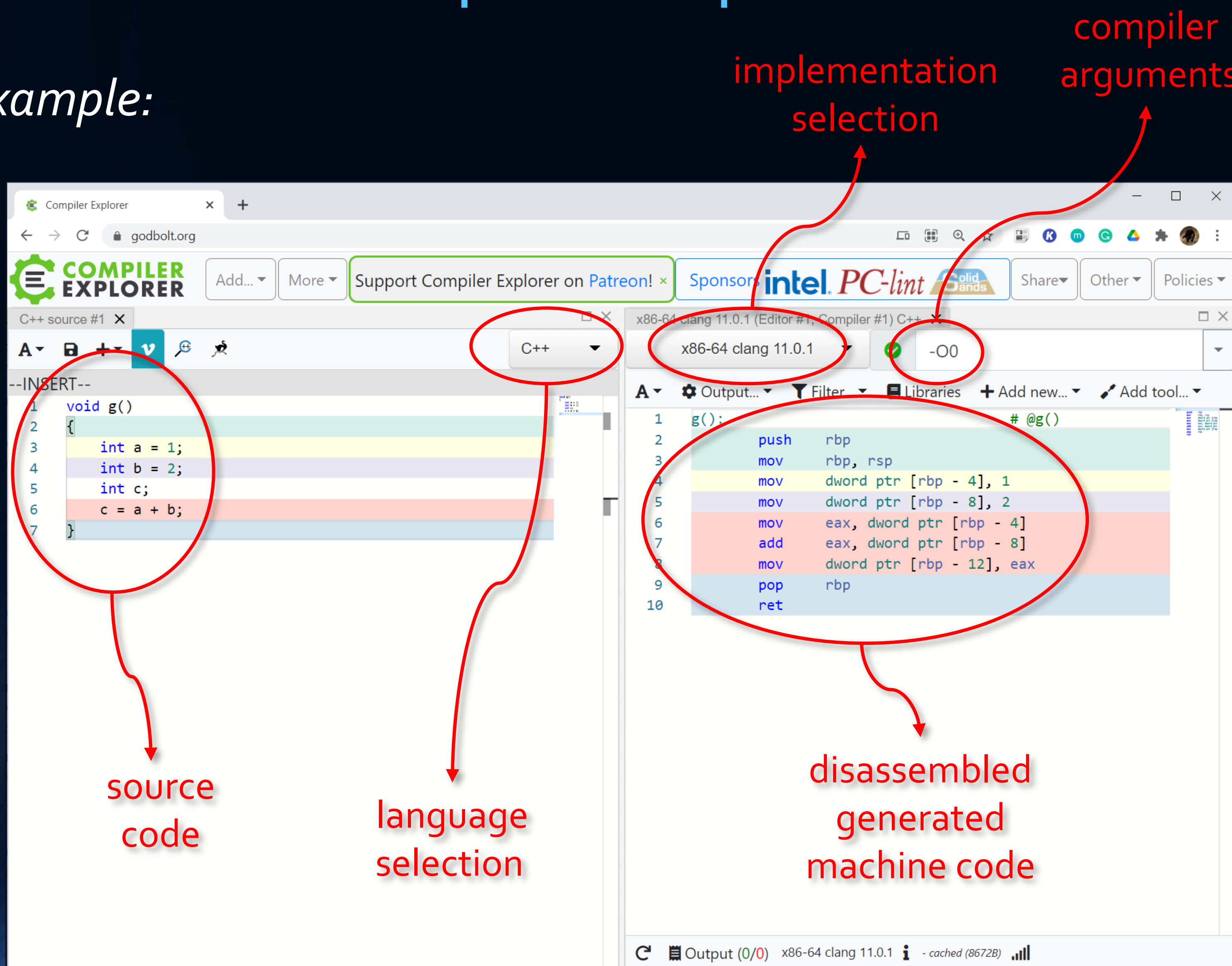
- *Recall:* observable behavior of function g = returning the control flow to the function caller.

Disassembling

- Reading machine code is hard for humans.
- Each machine code instruction (*opcode*) can be translated into a corresponding assembler instruction, which are much easier to read.
- This process is called **disassembling**.
- Special tools = disassemblers:
  - `objdump`, `gdb`, Compiler Explorer (*online*),…
- Compiler explorer (https://godbolt.org/):
  - excellent easy-to-use online disassembler;
  - many programming languages including C++;
  - many C++ implementations ("compilers") and their versions (*GCC, Clang, Intel, MSVC,…*);
  - multiple target architectures (*x86, x86_64, ARM64, MIPS64, power64, RISC-V,…*);
  - created and maintained by *Matt Godbolt*.

Compiler Explorer

- *Example:*



Optimizations *(cont.)*

- *Example (cont.)* — disassembled generated machine code with disabled optimizations:

```cpp
void g()
{
  int a = 1;
  int b = 2;
  int c;
  c = a + b;
  // implicit return
}
```

```asm
// clang++ -O0:
g():
    // initialize stack frame:
  push rbp
  mov  rbp, rsp
    // initialize a mapped to address (rbp-4):
  mov  dword ptr [rbp - 4], 1
    // initialize b mapped to address (rbp-8):
  mov  dword ptr [rbp - 8], 2
  mov  eax, dword ptr [rbp - 4]  // :load a from memory
    // load b from memory and add with a:
  add  eax, dword ptr [rbp - 8]
    // store result to c mapped to address (rbp-12):
  mov  dword ptr [rbp - 12], eax
  pop  rbp  // :restore stack frame
  ret       // :return control flow to caller
```

- *Observation:* machine code reflects all the source code, independently of whether it does or does not contribute to the observable behavior.
- *Live demo:* https://godbolt.org/z/6fGGEd.

Optimizations *(cont.)*

- *Example (cont.)* — disassembled generated machine code with enabled optimizations:

```
void g() {
  int a = 1;
  int b = 2;
  int c;
  c = a + b;
  // implicit return
}
```

```
// clang++ -O2:
g():
  ret  // :return control flow to caller
```

- *Observation:* machine code reflects only observable function behavior.
- *Live demo:* https://godbolt.org/z/aWx7c5.
- *Common C++ (or C) myth:* function-local (non-static) variables are stored on the stack → *false*.

- *Note:* CPUs don't know the concept of a variable ⇒ better considering variable *binary representation* being stored somewhere than variables themselves.
- *Truth:* Variable binary representation may be stored on the stack, somewhere else (e.g., in registers), nowhere at all.

Use case — benchmarking

- The as-if rule is highly relevant to writing benchmarks.
- **Benchmarking** — measuring runtime or performance of some operations.
- *Example:* measuring single-core floating-point performance on small (*fit-into-cache*) matrix multiplication:

```
void benchmark() {
  double A[32][32] = {};
  double B[32][32] = {};
  double C[32][32] = {};
  for (int n = 0; n < 1'000'000'000; n++)  // billion times...
    for (int i = 0; i < 32; i++)           // ... multiply C ← C + A * B
      for (int j = 0; j < 32; j++)
        for (int k = 0; k < 32; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```

$$C_{i,j} \leftarrow C_{i,j} + \sum_k A_{i,k} B_{k,j}$$

- Total number of FP operations: *N = 65.536 trillions*.
- *FP performance = measured function runtime / N*.
- Measurement on Microsoft Surface (i5, 2017): 40.96 EFlop/s.

Use case — benchmarking *(cont.)*

- *Microsoft Surface*, single core: 40.96 EFlop/s.
- *Fugaku supercomputer*, 7.6 million cores: 0.442 EFlop/s.
- *Apparent conclusion:* my Surface is 93× faster then the fastest supercomputer in the world (in Q1/2021).

- *Explanation:* observable behavior of benchmark function is only implicit return ⇒ measured number of FP operations = *0*.

```
void benchmark() {

  double A[32][32] = {};
  double B[32][32] = {};
  double C[32][32] = {};

  for (int n = 0; n < 1'000'000'000; n++)
    for (int i = 0; i < 32; i++)
      for (int j = 0; j < 32; j++)
        for (int k = 0; k < 32; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```

```
// g++ -O2:

benchmark():
  ret
```

- *Conclusion:* benchmarked operations must contribute to the observable behavior to avoid them being optimized away.

Machine code → source code?

- Can we obtain source code from which some machine code has been generated?

```
void benchmark() {

  double A[32][32] = {};
  double B[32][32] = {};
  double C[32][32] = {};

  for (int n = 0; n < 1'000'000'000; n++)
    for (int i = 0; i < 32; i++)
      for (int j = 0; j < 32; j++)
        for (int k = 0; k < 32; k++)
          C[i][j] += A[i][k] * B[k][j];
}
```

```
void g()
{
   int a = 1;
   int b = 2;
   int c;
   c = a + b;
}
```

```
void
empty(){ }
```

```
class Int
{
  int i_;
public:
  Int(int i) : i_(i) { }
  operator int&() { return i_; }
};

void G()
{
  Int a = 1;
  Int b = 2;
  int c;
  c = a + b;
}
```

```
ret
```

- 4 functions with very different source code but with same observable behavior ⇒ possibly same machine code (*live demo:* https://godbolt.org/z/hvvvYY).
- *Implication:* generally, it is not possible to "link" machine code back to source code.
- If needed, special mechanisms must be involved (*debugging information,…*).

Translation units

- Observable behavior is related to translation units.
- **Translation unit** — a piece of source code that is translated into machine code (compiled) at once.
- Basically: a translation unit = source file:
  - without comments
  - and processed by a C++ preprocessor.
- *Example:* source file (*left*), its translation unit with undefined `INT` (*right above*) and `INT` defined as `long` (*right below*):

```
// add.cpp source file

#ifndef INT
  #define INT int
#endif

// returns sum of its arguments:
INT add(INT a, INT b) {
  return a + b;
}
```

```
int add(int a, int b) {
  return a + b;
}
```

```
long add(long a, long b) {
  return a + b;
}
```

- *Note:* translation unit may be obtained by some implementation tools, such as „`g++ -DINT=long -E -P add.cpp`“.

Translation units *(cont.)*

- Compilers translate each translation unit separately.
- Machine code for a single translation unit is typically stored in so-called object file (`.o` or `.obj` file extension).
- To generate an object file with the machine code translated from a translation unit related to some source file, GCC or Clang uses:
  - `-c` command line option,
  - source file name as a command line argument.
- *Example:* `g++ -O2 -c add.cpp`

```
// add.cpp source file
#ifndef INT
  #define INT int
#endif
// returns sum of its arguments:
INT add(INT a, INT b) {
  return a + b;
}
```

source file
= source code

```
int add(int a, int b) {
  return a + b;
}
```

translation unit

```
add(int, int):
  8d 04 37 c3
```

object file
= machine code

```
add(int, int):
  lea eax, [rdi+rsi]
  ret
```

disassembled
object file

GCC/Clang workflow

- How do `g++` and `clang++` commands work?

a) *Input* = source file(s) and `-c` option ⇒ *translation/compilation*:

1. transforms the source file into a translation unit,
2. translates the source code of the translation unit into machine code,
3. stores that machine code in an object file.

- *Example:* `g++ -O2 -c main.cpp`.

b) *Input* = object file(s) ⇒ *linking*:

1. merges/links machine code from object file(s),
2. stores it into an executable binary file.

- *Note:* one of the object files must contain machine code of the `main` function.
- *Example:* `g++ main.o` (executable file is called `a.out` by default).

c) *Input* = source file(s), no `-c` option ⇒ *translation + linking*:

- *Ad a)* and *ad b)* at once.
- *Note:* an object file is not explicitly stored (it is stored only temporarily, for example in `/tmp` directory with some random file name).

GCC/Clang workflow *(cont.)*

- *Example* — source code:

```
// main.cpp
int main() {
  return 12;
}
```

- Separate compilation and linking:

```
$ g++ -O2 -c main.cpp
$ g++ main.o
$ ./a.out
$ echo $?
12
```

- Single-command alternative:

```
$ g++ -O2 main.cpp
$ ./a.out
$ echo $?
12
```

- *Observable behavior of* `main`*:* returns `12`.
- *C++ standard:* value returned from `main` = program exit status.
- *Bash:* exit status of terminated program — `$?` shell parameter.

Translation units *(cont.)*

- *Previous example:* only a single source file ⇒ a single translation unit.
- Source code of programs usually consists of multiple translation units.
- *Problems:*
  - Multiple translation units must cooperate at the source code level.
  - Machine code translated from multiple translation units must cooperate at the machine code level.
  - How is observable behavior resolved in the context of multiple translation units?
- *Example:*

```cpp
// add.cpp source file
int add(int a, int b) {
  return a + b;
}
```

```cpp
// main.cpp source file ver.1
int main() {
  return add(1, 2) + 3;
}
```

- In one source file, we want to use a function defined in another source file.

Translation units *(cont.)*

- Translation units for `add.cpp` and `main.cpp`, respectively:

```
int add(int a, int b) {
  return a + b;
}
```

```
int main() {
  return add(1, 2) + 3;
}
```

- Compilation of `add.cpp` is ok:

```
$ g++ -O2 -c add.cpp
```

- *Problem:* compilation of main.cpp is not ok:

```
$ g++ -O2 -c main.cpp
error: 'add' was not declared in this scope
```

- Observable behavior of `main`:
  - calling function named add with 1 and 2 integer arguments,
  - adding returned value with integer 3,
  - returning the result out of `main`.
- A compiler needs to generate machine code for this observable behavior. Why did it fail?

Translation units *(cont.)*

- During translation, a compiler always works only with a single translation unit.

```
int main() { return add(1, 2) + 3; }
```

- When a compiler translates `main.cpp`, what does it "see"?
  1. add function call,
  2. passed arguments (integer literals having values 1 and 2),
  3. what is done with the returned value (addition with 3).
- This is not enough for generating machine code for the observable behavior. Why?
  - For example, are the arguments passed to add by value or by reference?
  - Or, is there any implicit conversion required?
  - Such things can make difference at the machine code level.
- *Implication:* to be able to translate a function call, a compiler must see the type of its parameters as well as its return type.

Function declaration

- Type of function parameters and its return type makes part of its interface (API).
- In C++, function API **= function declaration** (a.k.a. *prototype*).
- Using (= calling) a function in some translation unit requires its declaration to be present in that translation unit as well.
- *Note:* function definition is also its declaration.
- Minimal declaration of add function:

```
int add(int, int);
```

- Addition into `main.cpp`:

```
// main.cpp source file ver.2
int add(int, int);
int main() { return add(1, 2) + 3; }
```

- Compilation is now ok:
  - types of parameters are known, return type is known.

```
$ g++ -O2 -c main.cpp
```

Translation units *(cont.)*

- API (such as declarations for functions) resolves cooperation of translation units at a source code level.
- What about the machine code level?
- Translation unit of `main.cpp`:

```
int add(int, int);
int main() { return add(1, 2) + 3; }
```

- Its compilation with…

```
$ g++ -O2 -c main.cpp
```

- …results in an object file `main.o` that contains machine code of function `main` that reflects its observable behavior.
- If we try to build an executable program file from it, it fails:

```
$ g++ main.o
undefined reference to `add(int, int)'
```

- *Why?*

Translation units *(cont.)*

- The part of the observable behavior of `main` is calling add.
- This observable behavior is reflected in its machine code:

```
main:
  sub  rsp, 8
  mov  esi, 2
  mov  edi, 1
  call add(int, int)
  add  rsp, 8
  add  eax, 3
  ret
```

- To build the executable binary, the linker needs to connect (link) this `call` instruction with the machine code of the add function.
- However, the linker cannot find this machine code.
- Machine code of add cannot be generated from the translation unit of `main.cpp` ⇒ it is not included in `main.o` object file.
- This problem triggers the "well-known" *undefined reference* error:

```
undefined reference to `add(int, int)'
```

Translation units *(cont.)*

- The only way the machine code of add can be generated is from its definition.
- **Function definition** basically = function declaration + its body.
- In our case, this definition is in the translation unit of `add.cpp` (*left*) ⇒ its compilation therefore generates machine code of add function in `add.o` object file (*right*):

```
int add(int a, int b) {
  return a + b;
}
```

```
add(int, int):
  lea eax, [rdi+rsi]
  ret
```

- To generate the final executable file, we need to provide both `main.o` and `add.o` object files to the linker:

```
$ g++ main.o add.o
```

ABI

```
$ g++ main.o add.o
```

- The final executable file `a.out` contains machine code of both functions main and add:

```
int add(int, int);

int main()
{
  return add(1, 2) + 3;
}
```

```
int add(int a, int b)
{
  return a + b;
}
```

- Both pieces of machine code need to cooperate at a machine code level.
- Why? Because of passing arguments from main to add and the return value from add back to main.

```
main:
  sub   rsp, 8
  mov   esi, 2
  mov   edi, 1
  call  add(int, int)
  add   rsp, 8
  add   eax, 3
  ret

add(int, int):
  lea eax, [rdi+rsi]
  ret
```

- The `call` instruction doesn't provide such a functionality (it only stores the current value of the *instruction pointer* on the stack and sets its new value to the address of the machine code of add).
- The arguments and the return value must be passed by some explicit mechanism.

ABI *(cont.)*

- In order to allow different functions to cooperate at a machine code level, there must be some agreement about how to pass arguments and return values to/from functions.
- This agreement (*calling conventions*) is a part of so-called **application binary interface (ABI)**.
- API:
  - enables cooperation of translation units at a source code (*programming*) level;
  - is defined by C++ standards;
  - *example*: function declaration allows to compile function calls.
- ABI:
  - enables cooperation of machine code generated from different translation units at a machine code (*binary*) level;
  - is not defined by C++ standards (typically, it is defined by the architecture + operating system + C++ implementation);
  - *example:* specification of how to pass arguments to functions.

ABI *(cont.)*

- In my case — x86_64 + Linux + GCC — the applied ABI specifies that:
  - `int` is a 32-bit signed integer;
  - argument for the first parameter of type `int` is passed into a function through the `edi` register;
  - argument for the second parameter of type `int` is passed into a function through the `esi` register;
  - return value of type `int` is passed from a function through the `eax` register.

```
int main()
{
  return add(1, 2) + 3;
}
```

```
main:
  sub  rsp, 8
  mov  esi, 2
  mov  edi, 1
  call add(int, int)
  add  rsp, 8
  add  eax, 3
  ret
```

```
add(int, int):
  lea eax, [rdi+rsi]
  ret
```

  - *Note:* the latter holds for `add` as well as for `main`.
  - *Note:* the ABI requires stack frames to be 16-byte aligned ⇒ `rsp` register manipulations.

ABI *(cont.)*

- Importance of ABI — it allows linking the machine code generated:
  - by different C++ implementations,
  - at different places (different computers),
  - at different times.

```
$ clang++ -O2 -c add.cpp
$ g++ -O2 -c main.cpp
$ g++ main.o add.o
$ ./a.out
$ echo $?
6
```

- ABI is a must for using libraries.
- **Library:**
  - A collection of object files that implements some functionality bundled in a file with specific format (`.a`, `.so`, `.lib`, `.dll`,…).
  - Plus API (usually *header files*) that enables to work with that functionality at a source code level (for example, library function declarations).