Compilation Process

Ivan Sorokin
Сорокин Иван Владимирович

Introduction

- In this lecture we'll discuss how programs consisting of multiple files are built.
- Normally this topic is not covered well in literature:
  - For books for beginners this topic is considered too advanced.
  - Advanced books assume that the reader already familiar with the topic.
- Still students often face related problems:
  - The compiler shows an absolutely correct error message, but because the student doesn't have a mental model of how the program is built, he doesn't know what is needed to fix it.
- The questions related to program compilation are a good fit for the exam:
  - If you don't know — it is impossible to guess.
- Even real and big programs have defects like the ones that we'll discuss today.

g++ command

The command g++ is commonly used to compile programs.

```
// hello.cpp
#include <stdio.h>

int main()
{
    puts("Hello, world!");
}
```

```
$ g++ hello.cpp
$ ./a.out
Hello world!
```

g++ command

The **g++** command doesn't translate the program on its own. Instead it just runs other commands.

One can use *strace -f -e trace=execve* to see what really is run.

```
$ g++ hello.cpp
    cc1plus ... hello.cpp -o hello.s
    as hello.s -o hello.o
    collect2 hello.o -o hello
        ld hello.o -o hello
```

Step-by-step compilation (1)

```
$ cat hello.cpp
#include <stdio.h>

int main()
{
    puts("Hello, world!");
}
```

Step-by-step compilation (2)

```
$ g++ -E -P hello.cpp > hello.i
$ cat hello.i
extern "C" {
typedef long unsigned int size_t;
typedef __builtin_va_list
__gnuc_va_list;
// ... 330 lines ...
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
}
int main()
{
    puts("Hello, world!");
}
```

- The preprocessing is the first step of the compilation.

  Preprocessor is done by the command `g++ -E`

- It handles all things that start with **#** (so-called preprocessor directives).
  - For example the preprocessor expands the **`#include`** directive to the text of the file.

Step-by-step compilation (3)

```
$ g++ -S -masm=intel -O2 hello.i
$ cat hello.s
        .text
        .section        .rodata
.LC0:
        .string "Hello, world!"
        .section        .text
        .globl  main
main:
        sub     rsp, 8
        lea     rdi, .LC0[rip]
        call    puts@PLT
        xor     eax, eax
        add     rsp, 8
        ret
```

- The second step of the compilation is called translation.

  The translation is done by the command `g++ -S`

- It translates the text of the C++ program to the assembly language.

Step-by-step compilation (4)

```
$ as -o hello.o hello.s
$ xxd hello.o
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.............
00000020: 0000 0000 0000 0000 3002 0000 0000 0000  ........0.......
00000030: 0000 0000 4000 0000 0000 4000 0e00 0d00  ....@.....@.....
00000040: 4865 6c6c 6f2c 2077 6f72 6c64 2100 0000  Hello, world!...
00000050: 4883 ec08 488d 3d00 0000 00e8 0000 0000  H...H.=.........
00000060: 31c0 4883 c408 c300 4743 433a 2028 5562  1.H.....GCC: (Ub
00000070: 756e 7475 2031 312e 342e 302d 3175 6275  untu 11.4.0-1ubu
00000080: 6e74 7531 7e32 322e 3034 2920 3131 2e34  ntu1~22.04) 11.4
00000090: 2e30 0000 0000 0000 1400 0000 0000 0000  .0..............
000000a0: 017a 5200 0178 1001 1b0c 0708 9001 0000  .zR..x..........
000000b0: 1400 0000 1c00 0000 0000 0000 1700 0000  ................
000000c0: 0044 0e10 520e 0800 0000 0000 0000 0000  .D..R...........
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ..............…
...
```

- The third step of the compilation is called assembling.

  The translation is done by the command *as*

- It translates the assembly program to the machine code.
  - The result of assembly is a binary file called object file.

Step-by-step compilation (5)

```
$ objdump -drC -Mintel hello.o
hello.o:     file format elf64-x86-64

Disassembly of section .text.startup:

0000000000000000 <main>:
   0:   48 83 ec 08             sub    rsp,0x8
   4:   48 8d 3d 00 00 00 00    lea    rdi,[rip+0x0]        # b <main+0xb>
                        7: R_X86_64_PC32        .LC0-0x4
   b:   e8 00 00 00 00          call   10 <main+0x10>
                        c: R_X86_64_PLT32       puts-0x4
  10:   31 c0                   xor    eax,eax
  12:   48 83 c4 08             add    rsp,0x8
  16:   c3                      ret
```

- The content of the object file can be examined by a command **`objdump`**.

Step-by-step compilation (6)

```
$ g++ -o hello hello.o
$ ./hello
Hello, world!
```

- Finally the fourth and the last step of the compilation is linking.
  - It takes the object files and produces the final executable binary.

Object file vs executable files

- Both are binary
- Object files are designed to be an input for the linker. Executable files are designed to load and run efficiently.
  - On Windows they have different formats: COFF for object files and PE for executables.
  - Still they have lots in common and on Linux the same file format (ELF) is used for both. Still some content is specific for each:
    - Object files are normally relocatable. Executable files are commonly (but not always) non-relocatable.
    - In executable files the linker inserts data for dynamic library loading like GOT and PLT. Object files don’t need them.

Multiple Translation Units (1)

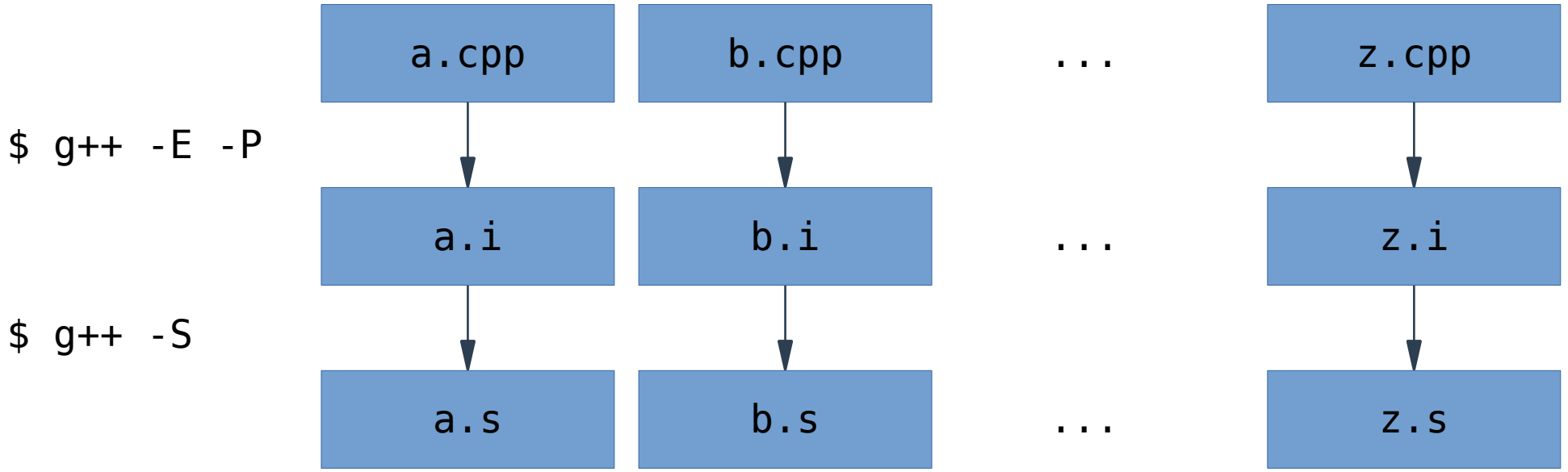
- All the information above was about the programs consisting of one file.
- Let us have a look at what the compilation of programs consisting of multiple files looks like.

Multiple Translation Units (2)

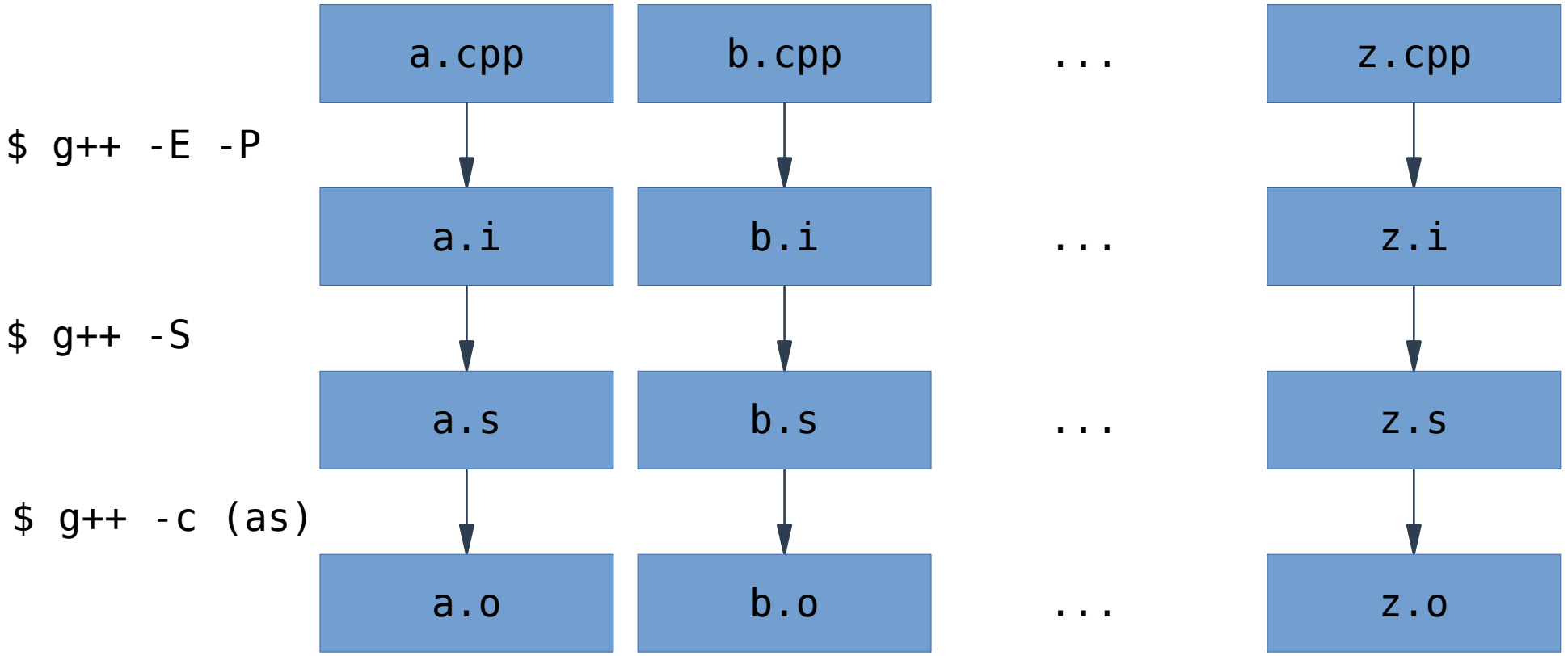
Multiple Translation Units (3)

```
$ g++ -E -P
```

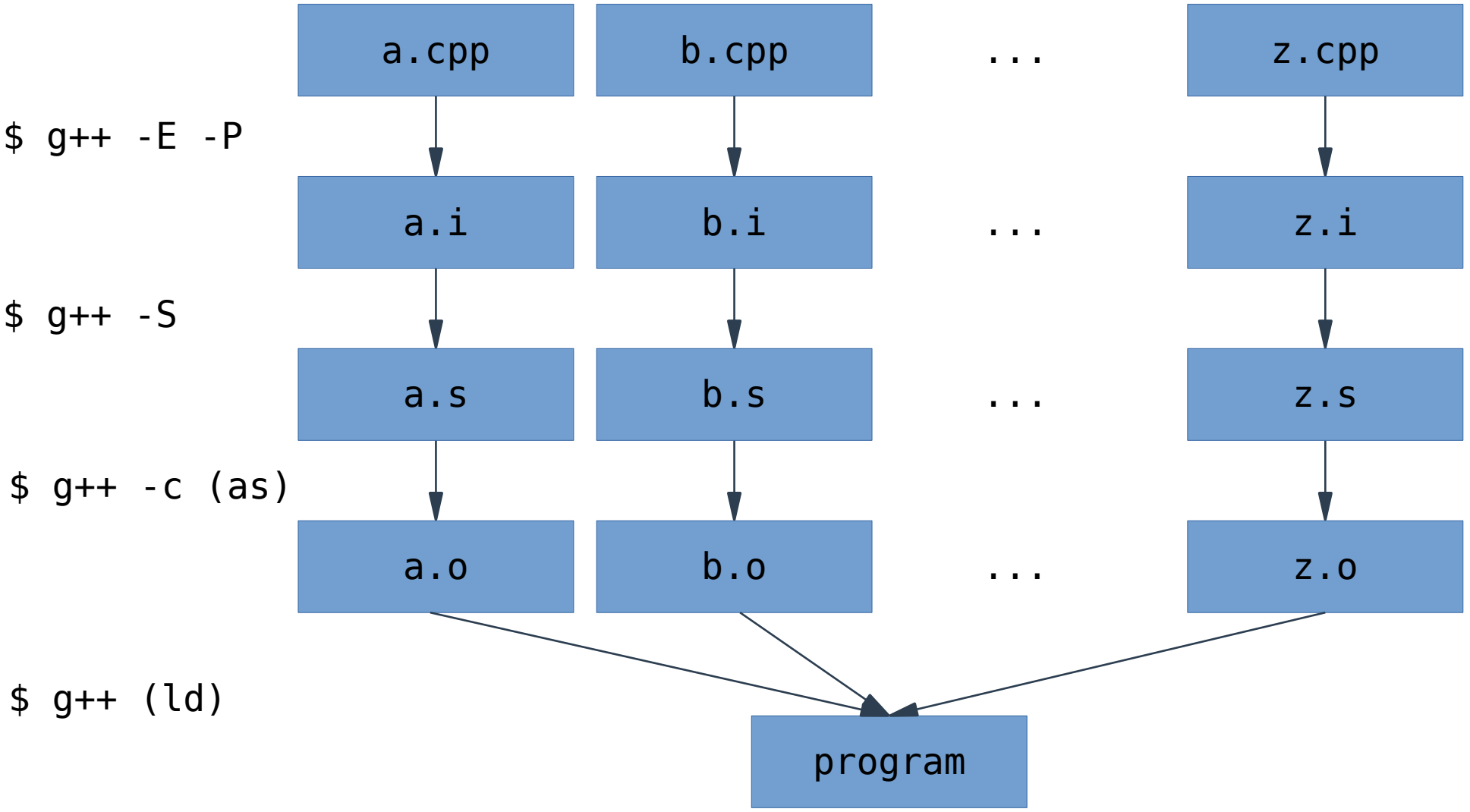
Multiple Translation Units (4)



Multiple Translation Units (5)



Multiple Translation Units (6)



Multiple Translation Units (7)

- The preprocessing, the translation, the assembling are done independently for each **`.cpp`** file.
  - The **`.cpp`** files participating in the compilation are called *translation units*.
- Then all the objects files are linked together to produce the program executable.
  - This model allows incremental compilation. When the source is changed only the modified translation units need to be preprocessed, translated and assembled. For the rest the object files from the previous compilation can be reused.
  - Also this model uses memory conservatively because none of the steps requires the program to be loaded in memory fully.

Declaration and definitions (1)

Let us compile a program with multiple translation units:

```
// a.cpp
int main()
{
    foo(42);
}
```

```
// b.cpp
#include <cstdint>
#include <cstdio>

void foo(int32_t a)
{
    printf("Answer: %d\n", a);
}
```

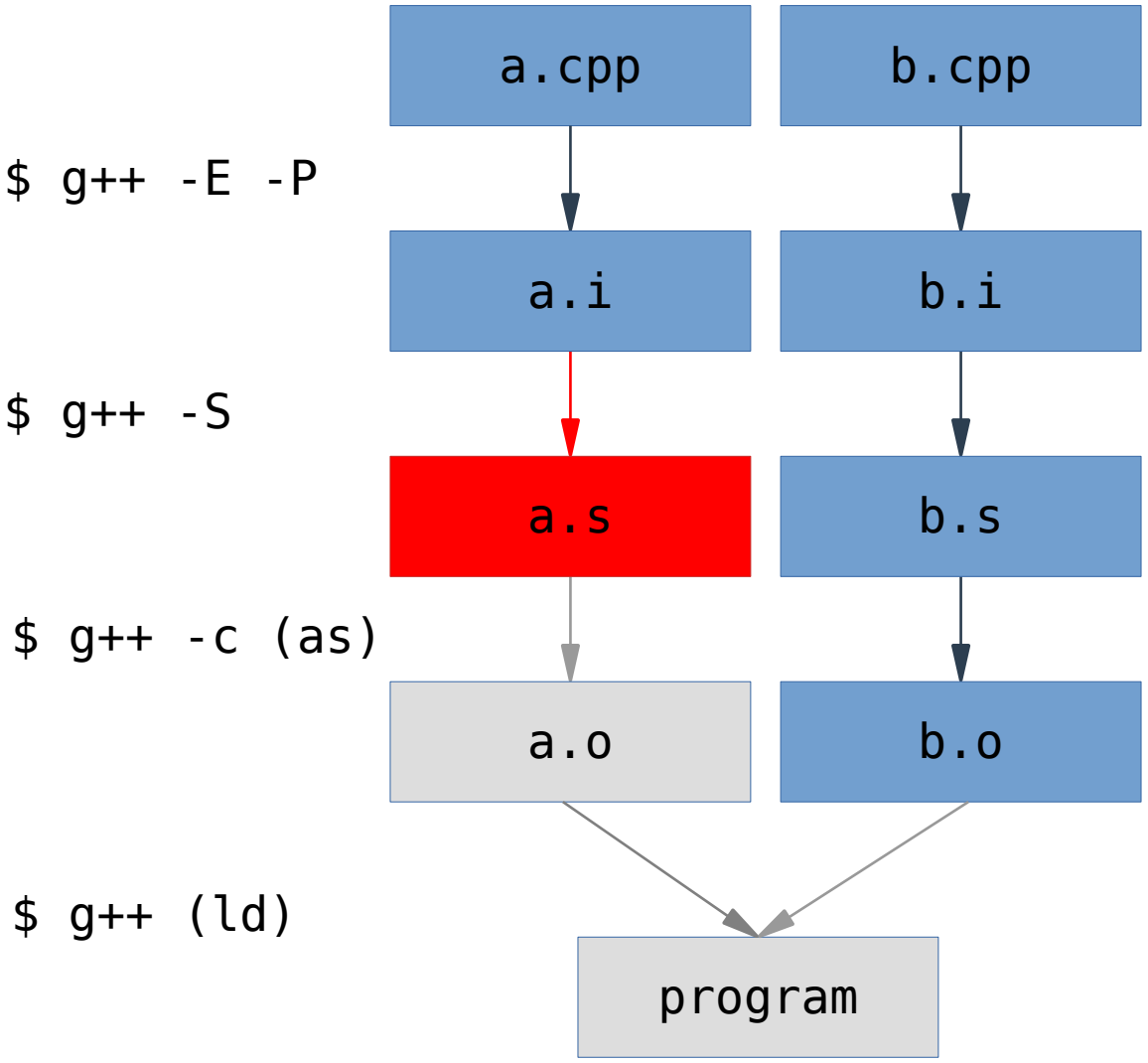
Declaration and definitions (2)

```
// a.cpp
int main()
{
    foo(42);
}
```

```
// b.cpp
#include <cstdint>
#include <cstdio>

void foo(int32_t a)
{
    printf("Answer: %d\n", a);
}
```

```
$ g++ a.cpp b.cpp
a.cpp: In function ‘int main()’:
a.cpp:4:5: error: ‘foo’ was not declared in this scope
    4 |     foo(42);
      |     ^~~
```

Declaration and definitions (3)




```
// a.cpp
int main()
{
    foo(42);
}
```

```
$ g++ -S a.cpp
a.cpp: In function ‘int main()’:
a.cpp:4:5: error: ‘foo’ was not declared in this scope
    4 |     foo(42);
      |     ^~~
```

Declaration and definitions (4)

```
// a.cpp
#include <cstdint>

void foo(int32_t);

int main()
{
    foo(42);
}
```

```
// b.cpp
#include <cstdint>
#include <cstdio>

void foo(int32_t a)
{
    printf("Answer: %d\n", a);
}
```

```
$ g++ a.cpp b.cpp
$ ./a.out
Answer: 42
```

Declaration and definitions (5)

```
#include <cstdint>
void foo(int32_t);
int main()
{
    foo(42);
}
```

```
main:
        sub     rsp, 8
        mov     edi, 42
        call    _Z3fooi
        xor     eax, eax
        add     rsp, 8
        ret
```

```
#include <cstdint>
void foo(float);
int main()
{
    foo(42);
}
```

```
main:
        sub     rsp, 8
        movss   xmm0, DWORD PTR .LC0[rip]
        call    _Z3foof
        xor     eax, eax
        add     rsp, 8
        ret
.LC0:
        .long   1109917696
```

Declaration and definitions (6)

```
void foo(int32_t);
```

Function declaration means the function with this signature exists somewhere in the program.

- Usually in other translation unit.
- Can be in the same translation unit below.

```
void foo(int32_t) {}
```

Function definition means the function has this body.

Declaration and definitions (7)

```
// a.cpp
#include <cstdint>

void foo(int32_t);

int main()
{
    foo(42);
}
```

```
// b.cpp
#include <cstdint>
#include <cstdio>

void foo(int32_t a)
{
    printf("Answer: %d\n", a);
}
```

```
$ g++ a.cpp
```

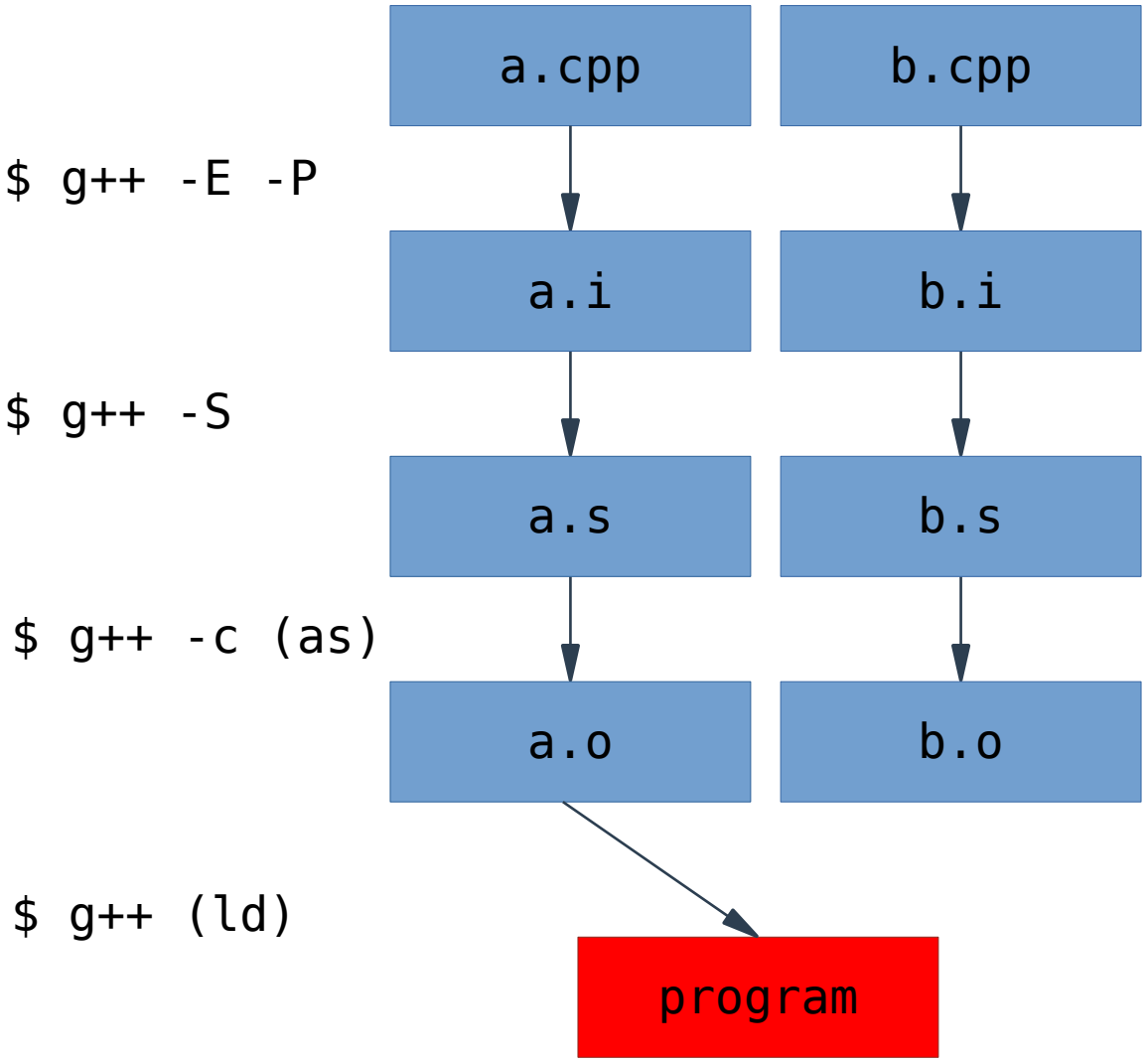
Declaration and definitions (8)

```
// a.cpp
#include <cstdint>

void foo(int32_t);

int main()
{
    foo(42);
}
```

```
// b.cpp
#include <cstdint>
#include <cstdio>

void foo(int32_t a)
{
    printf("Answer: %d\n", a);
}
```

```
$ g++ a.cpp
/usr/bin/ld: /tmp/ccIzCI7K.o: in function `main':
a.cpp:(.text+0xe): undefined reference to `foo(int)'
collect2: error: ld returned 1 exit status
```

Declaration and definitions (9)




```
// a.cpp
#include <cstdint>

void foo(int32_t);

int main()
{
    foo(42);
}
```

```
$ g++ a.cpp
/usr/bin/ld: /tmp/ccIzCI7K.o: in function `main':
a.cpp:(.text+0xe): undefined reference to
`foo(int)'
collect2: error: ld returned 1 exit status
```

Declaration and definitions (10)

```
// a.cpp
#include <cstdint>

void foo(int32_t);

int main()
{
    foo(42);
}
```

```
// b.cpp
#include <cstdint>
#include <cstdio>

void foo(int32_t a)
{
    printf("Answer: %d\n", a);
}
```

```
$ g++ b.cpp
```

Declaration and definitions (11)

```
// a.cpp
#include <cstdint>

void foo(int32_t);

int main()
{
    foo(42);
}
```

```
// b.cpp
#include <cstdint>
#include <cstdio>

void foo(int32_t a)
{
    printf("Answer: %d\n", a);
}
```

```
$ g++ b.cpp
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o: in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Declaration and definitions (12)

- The function _start is the first function that is executed the the program is run.
  - https://elixir.bootlin.com/glibc/glibc-2.39/source/sysdeps/x86_64/start.S

```
ENTRY (_start)
        cfi_undefined (rip)
        xorl %ebp, %ebp
...
        mov main@GOTPCREL(%rip), %RDI_LP
        call *__libc_start_main@GOTPCREL(%rip)
        hlt
END (_start)
```

Declaration and definitions (13)

Function declaration are needed not only to call functions from other translation units, but also to use the function that that are defined in the same translation unit, but below.

```
#include <cstdio>

int main()
{
    f();
}

void f()
{
    printf("Hello, world!");
}
```

```
$ g++ a.cpp
a.cpp: In function 'int main()':
a.cpp:5:5: error: 'f' was not declared in this scope
    5 |     f();
      |     ^
```

Declaration and definitions (14)

```
#include <cstdio>

void f()
{
    printf(“Hello, world!”);
}

int main()
{
    f();
}
```

```
#include <cstdio>

void f();

int main()
{
    f();
}

void f()
{
    printf(“Hello, world!”);
}
```

Declaration and definitions (15)

For regular functions reordering helps, but mutually recursive functions require a declaration.

```
#include <cstdint>
#include <iostream>

void f(uint32_t i);

void g(uint32_t i)
{
    std::cout << i << ' ';
    f(i);
}

void f(uint32_t i)
{
    if (i == 100)
      return;

    g(i + 1);
}

int main()
{
    f(0);
}
```

Multiple definitions (1)

```
// main.cpp

void hello();

int main()
{
    hello();
}
```

```
// a.cpp

#include <cstdio>

void hello()
{
    puts("Hello, a.cpp!");
}
```

```
// b.cpp

#include <cstdio>

void hello()
{
    puts("Hello, b.cpp!");
}
```

```
$ g++ -g main.cpp a.cpp b.cpp
```

Multiple definitions (2)

```
// main.cpp

void hello();

int main()
{
    hello();
}
```

```
// a.cpp

#include <cstdio>

void hello()
{
    puts("Hello, a.cpp!");
}
```

```
// b.cpp

#include <cstdio>

void hello()
{
    puts("Hello, b.cpp!");
}
```

```
$ g++ -g main.cpp a.cpp b.cpp
/usr/bin/ld: /tmp/cc76XtFk.o: in function `hello()':
b.cpp:4: multiple definition of `hello()'; /tmp/cc7jY98q.o:a.cpp:4:
first defined here
collect2: error: ld returned 1 exit status
```

Multiple definitions (2)

```
// main.cpp

void hello();

int main()
{
    hello();
}
```

```
// a.cpp

#include <cstdio>

void hello()
{
    puts("Hello, a.cpp!");
}
```

```
// b.cpp

#include <cstdio>

void hello()
{
    puts("Hello, b.cpp!");
}
```

```
$ g++ -g main.cpp a.cpp b.cpp
b.cpp:4: multiple definition of `hello()'
```

```
$ g++ -g main.cpp a.cpp
$ ./a.out
Hello, a.cpp!
```

```
$ g++ -g main.cpp b.cpp
$ ./a.out
Hello, b.cpp!
```

```
$ g++ -g main.cpp
main.cpp:5: undefined reference to `hello()'
```

Multiple definitions (3)

```
// main.cpp

void hello();

int main()
{
    hello();
}
```

```
// a.cpp

#include <cstdio>

void hello()
{
    puts("Hello, a.cpp!");
}
```

```
// b.cpp

#include <cstdio>

void hello()
{
    puts("Hello, b.cpp!");
}
```

```
$ nm -C main.o
                 U hello()
0000000000000000 T main

$ nm -C a.o
0000000000000000 T hello()
                 U puts

$ nm -C b.o
0000000000000000 T hello()
                 U puts
```

static declaration specifier (1)

In order to make function local to the translation unit one can use the declaration specifier **`static`**.

Static functions are said to have *internal linkage* (local to translation unit).

Non-static functions are said to have *external linkage* (accessible through the whole program).

```
#include <cstdio>

static void hello()
{
    puts("Hello, world!");
}

int main()
{
    hello();
}
```

static declaration specifier (2)

```
// main.cpp

void hello();

int main()
{
    hello();
}
```

```
// a.cpp

#include <cstdio>

static void hello()
{
    puts("Hello, a.cpp!");
}
```

```
// b.cpp

#include <cstdio>

void hello()
{
    puts("Hello, b.cpp!");
}
```

```
$ g++ -g main.cpp a.cpp b.cpp
$ ./a.out
Hello, b.cpp!

$ nm -C main.o
                 U hello()
0000000000000000 T main

$ nm -C a.o
0000000000000000 t hello()
                 U puts

$ nm -C b.o
0000000000000000 T hello()
                 U puts
```

static declaration specifier (3)

Please note that static has 3 different meanings:

- On global functions: the function is local to the translation unit.
- On member functions: the function doesn't take parameter this.
- On local variables: the variable outlive the scope it is declared in and lives until the end of the program.

```
#include <cstdio>

static void hello()
{
    puts("Hello, world!");
}

int main()
{
    hello();
}
```

```
struct mytype
{
    void non_static_member();
    static void static_member();
};

void test(mytype& x)
{
    x.non_static_member();
    mytype::static_member();
}
```

```
int32_t next()
{
    static int32_t value = 0;
    return value++;
}
```

Variables (1)

Similar to functions variables also have declarations and definitions.

Declaration:

```
void hello();
```

Definition:

```
void hello()
{}
```

```
int x;
```

Variables (2)

Similar to functions variables also have declarations and definitions.

Declaration:

```
void hello();
```

Definition:

```
void hello()
{}

int x;
```

Variables (3)

Similar to functions variables also have declarations and definitions.

Declaration:

```
void hello();

extern int x;
```

Definition:

```
void hello()
{}

int x;
```

Variables (4)

```
// a.cpp
#include <cstdint>

extern int32_t a;

void f();

int main()
{
    f();
    a = 5;
    f();
}
```

```
// b.cpp
#include <cstdint>
#include <cstdio>

int32_t a;

void f()
{
    printf("%d\n", a);
}
```

```
$ g++ a.cpp b.cpp
```

Variables (5)

```
// a.cpp
#include <cstdint>

extern int32_t a;

void f();

int main()
{
    f();
    a = 5;
    f();
}
```

```
// b.cpp
#include <cstdint>
#include <cstdio>

int32_t a;

void f()
{
    printf("%d\n", a);
}
```

```
$ g++ a.cpp b.cpp
$ ./a.out
```

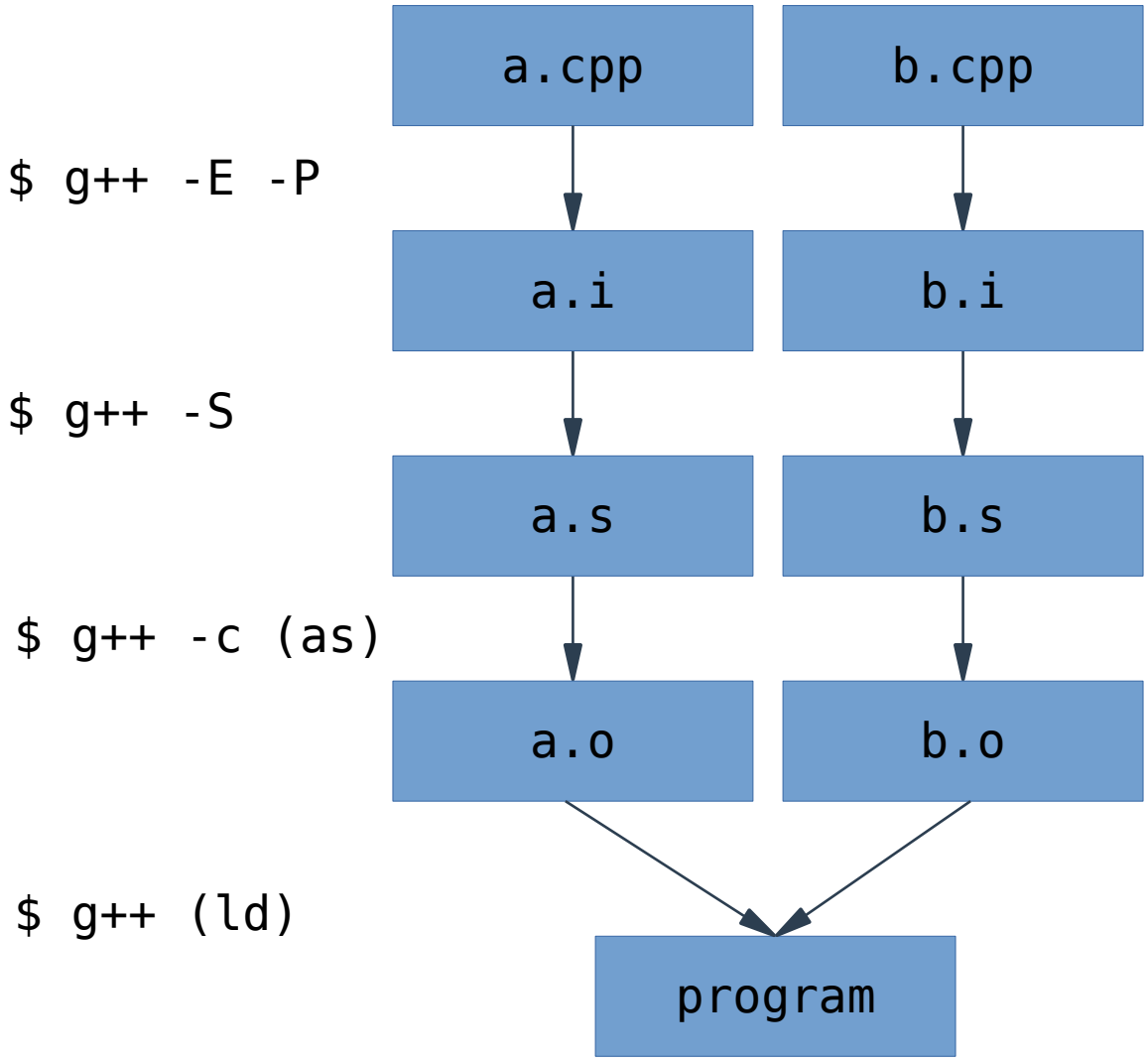
Variables (6)

```
// a.cpp
#include <cstdint>

extern int32_t a;

void f();

int main()
{
    f();
    a = 5;
    f();
}
```

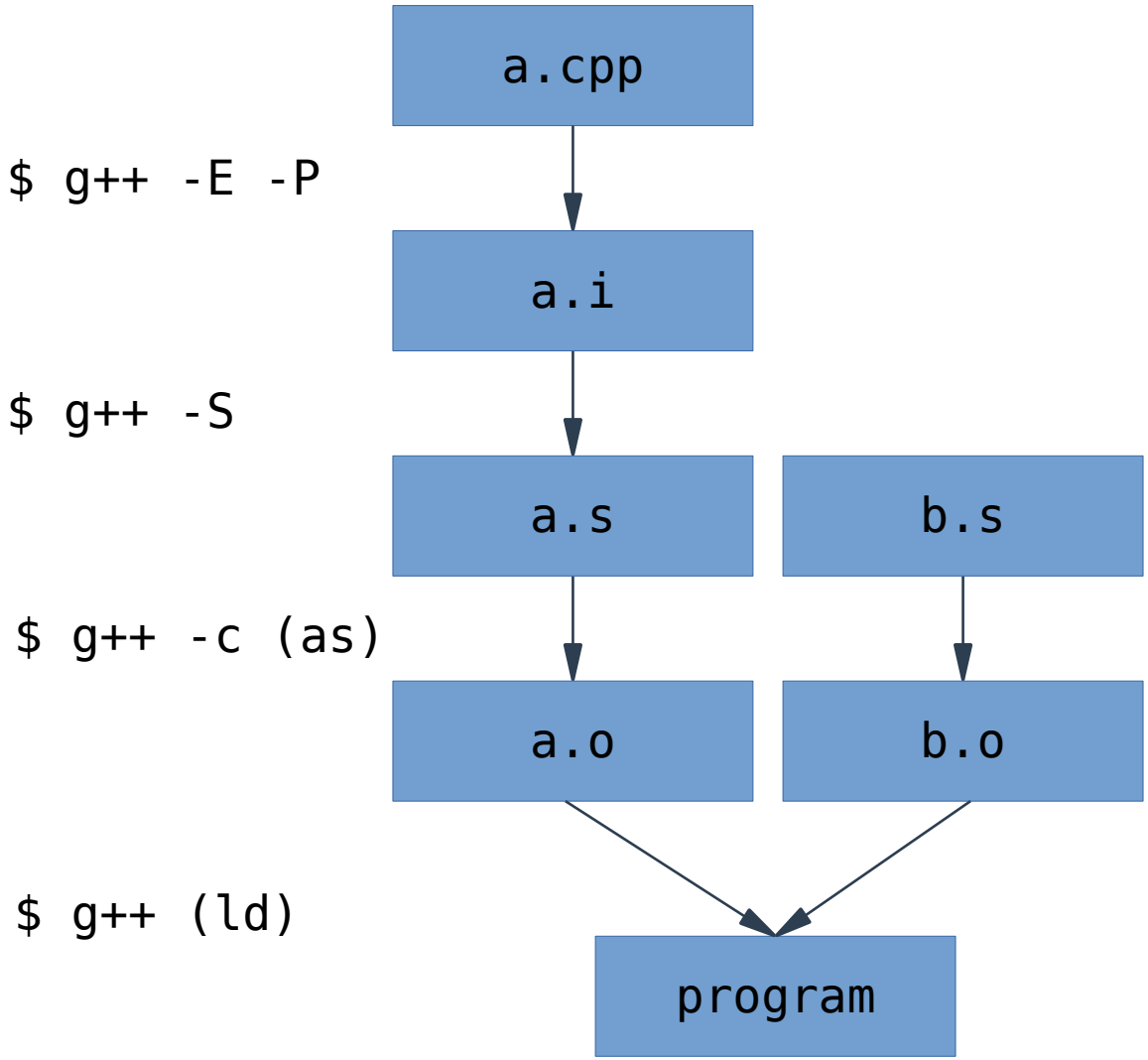
```
// b.cpp
#include <cstdint>
#include <cstdio>

int32_t a;

void f()
{
    printf("%d\n", a);
}
```

```
$ g++ a.cpp b.cpp
$ ./a.out
0
5
```

Assembly (1)



Assembly (2)



Assembly (3)

```
// main.cpp
#include <cstdint>
#include <cstdio>

extern "C" uint32_t fibonacci(uint32_t n);

int main()
{
    for (uint32_t i = 0; i != 12; ++i)
        printf("%d: %d\n", i, fibonacci(i));
}
```

```
# fibonacci.s
        .intel_syntax noprefix
        .text
        .globl      fibonacci

fibonacci:
        test        edi, edi
        je          .return_zero

        xor         eax, eax
        mov         ecx, 1

.loop:
        lea         esi, [rax + rcx]
        mov         eax, ecx
        mov         ecx, esi
        dec         edi
        jnz         .loop
        ret

.return_zero:
        xor         eax, eax
        ret
```

```
$ g++ main.cpp fibonacci.s
$ ./a.out
0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34
10: 55
11: 89
```

Assembly (4)

```
// main.cpp
#include <cstdint>
#include <cstdio>

uint32_t fibonacci(uint32_t n);

int main()
{
    for (uint32_t i = 0; i != 12; ++i)
        printf("%d: %d\n", i, fibonacci(i));
}
```

```
# fibonacci.s
        .intel_syntax noprefix
        .text
        .globl      fibonacci

fibonacci:
        test        edi, edi
        je          .return_zero

        xor         eax, eax
        mov         ecx, 1

.loop:
        lea         esi, [rax + rcx]
        mov         eax, ecx
        mov         ecx, esi
        dec         edi
        jnz         .loop
        ret

.return_zero:
        xor         eax, eax
        ret
```

```
$ g++ main.cpp fibonacci.s
/usr/bin/ld: /tmp/ccOFDNBs.o: in function `main':
main.cpp:(.text+0x1b): undefined reference to
`fibonacci(unsigned int)'
collect2: error: ld returned 1 exit status
```

Assembly (5)

```
// main.cpp
#include <cstdint>
#include <cstdio>

uint32_t fibonacci(uint32_t n);

int main()
{
    for (uint32_t i = 0; i != 12; ++i)
        printf("%d: %d\n", i, fibonacci(i));
}
```

```
$ nm -C main.o
                 U fibonacci(unsigned int)
0000000000000000 T main
                 U printf
$ nm -C fibonacci.o
000000000000000b t .loop
0000000000000017 t .return_zero
0000000000000000 T fibonacci
```

```
# fibonacci.s
        .intel_syntax noprefix
        .text
        .globl      fibonacci

fibonacci:
        test        edi, edi
        je          .return_zero

        xor         eax, eax
        mov         ecx, 1

.loop:
        lea         esi, [rax + rcx]
        mov         eax, ecx
        mov         ecx, esi
        dec         edi
        jnz         .loop
        ret

.return_zero:
        xor         eax, eax
        ret
```

Assembly (6)

```
// main.cpp
#include <cstdint>
#include <cstdio>

uint32_t fibonacci(uint32_t n);

int main()
{
    for (uint32_t i = 0; i != 12; ++i)
        printf("%d: %d\n", i, fibonacci(i));
}
```

```
$ nm main.o
                 U _Z9fibonaccij
0000000000000000 T main
                 U printf
$ nm fibonacci.o
000000000000000b t .loop
0000000000000017 t .return_zero
0000000000000000 T fibonacci
```

```
# fibonacci.s
        .intel_syntax noprefix
        .text
        .globl      fibonacci

fibonacci:
        test        edi, edi
        je          .return_zero

        xor         eax, eax
        mov         ecx, 1

.loop:
        lea         esi, [rax + rcx]
        mov         eax, ecx
        mov         ecx, esi
        dec         edi
        jnz         .loop
        ret

.return_zero:
        xor         eax, eax
        ret
```

Assembly (7)

```
// main.cpp
#include <cstdint>
#include <cstdio>

uint32_t fibonacci(uint32_t n);

int main()
{
    for (uint32_t i = 0; i != 12; ++i)
        printf("%d: %d\n", i, fibonacci(i));
}
```

```
# fibonacci.s
        .intel_syntax noprefix
        .text
        .globl      _Z9fibonaccij

_Z9fibonaccij:
        test        edi, edi
        je          .return_zero

        xor         eax, eax
        mov         ecx, 1

.loop:
        lea         esi, [rax + rcx]
        mov         eax, ecx
        mov         ecx, esi
        dec         edi
        jnz         .loop
        ret

.return_zero:
        xor         eax, eax
        ret
```

```
$ g++ main.cpp fibonacci.s
$ ./a.out
0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34
10: 55
11: 89
```

Name mangling (1)

C++ allows function overloading, thus symbol name includes the parameters types of the function.

Please note that GCC doesn't encode the return type in the symbol name.

```
// test.cpp
void test() {}
void test(int) {}
void test(unsigned) {}
void test(double) {}
void test(char const*) {}
```

```
$ g++ -c test.cpp
$ nm test.o
0000000000000000 T _Z4testv
000000000000000b T _Z4testi
0000000000000019 T _Z4testj
0000000000000027 T _Z4testd
0000000000000037 T _Z4testPKc
```

Name mangling (2)

One can use **`c++filt`** command to demangle a name.

```
$ c++filt _Z4testiii
test(int, int, int)
```

```
$ nm 1.o | c++filt
0000000000000000 T test()
000000000000000b T test(int)
0000000000000019 T test(unsigned int)
0000000000000027 T test(double)
0000000000000037 T test(char const*)
```

Some commands have built-in demangler:

- **`nm -C`**
- **`objdump -C`**

Name mangling (3)

The header **<stdio.h>** allows include both from C and C++.

```
$ cat hello.cpp
#include <stdio.h>

int main()
{
    puts("Hello, world!");
}
```

```
$ g++ -E -P hello.cpp > hello.i
$ cat hello.i
extern "C" {
typedef long unsigned int size_t;
typedef __builtin_va_list __gnuc_va_list;
// ... 330 lines ...
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
}
int main()
{
    puts("Hello, world!");
}
```

Name mangling (4)

```
$ cat /usr/include/stdio.h
...
#ifndef _STDIO_H
#define _STDIO_H        1

#define __GLIBC_INTERNAL_STARTING_HEADER_IMPLEMENTATION
#include <bits/libc-header-start.h>

__BEGIN_DECLS
...
__END_DECLS

#endif /* <stdio.h> included.  */



$ cat /usr/include/x86_64-linux-gnu/sys/cdefs.h
...
/* C++ needs to know that types and declarations are C, not C++.  */
#ifdef  __cplusplus
# define __BEGIN_DECLS  extern "C" {
# define __END_DECLS    }
#else
# define __BEGIN_DECLS
# define __END_DECLS
#endif
...
```

Libraries (1)

```
// test.cpp
extern "C" int printf(char const*, ...);

int main()
{
    printf("Hello, world\n");
}
```

```
$ g++ test.cpp
```

Libraries (2)

```
// test.cpp
extern "C" int printf(char const*, ...);

int main()
{
    printf("Hello, world!\n");
}
```

```
$ g++ test.cpp
$ ./a.out
Hello, world!
```

Libraries (3)

```
$ g++ -v test.o
...
 /usr/lib/gcc/x86_64-linux-gnu/11/collect2 -plugin
/usr/lib/gcc/x86_64-linux-gnu/11/liblto_plugin.so
-plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/11/lto-wrapper
-plugin-opt=-fresolution=/tmp/cc0iAT08.res -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-
pass-through=-lgcc -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc_s -plugin-
opt=-pass-through=-lgcc --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed
-dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now -z relro /usr/lib/gcc/x86_64-linux-
gnu/11/../../../x86_64-linux-gnu/Scrt1.o /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-
linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o -L/usr/lib/gcc/x86_64-linux-
gnu/11 -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-
linux-gnu/11/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-
gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/11/../../.. /tmp/ccLdsUU3.o -lstdc++ -
lm -lgcc_s -lgcc -lc -lgcc_s -lgcc /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (4)

```
ld
     -plugin /usr/lib/gcc/x86_64-linux-gnu/11/liblto_plugin.so
     -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/11/lto-wrapper
     -plugin-opt=-fresolution=/tmp/cc0iAT08.res
     -plugin-opt=-pass-through=-lgcc_s
     -plugin-opt=-pass-through=-lgcc
     -plugin-opt=-pass-through=-lc
     -plugin-opt=-pass-through=-lgcc_s
     -plugin-opt=-pass-through=-lgcc
     --build-id
     --eh-frame-hdr
     -m elf_x86_64
     --hash-style=gnu
     --as-needed
     -dynamic-linker /lib64/ld-linux-x86-64.so.2
     -pie
     -z now
     -z relro
     /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o
     /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crti.o
     /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
     -L/usr/lib/gcc/x86_64-linux-gnu/11
     -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu
     -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../../lib
     -L/lib/x86_64-linux-gnu -L/lib/../lib
     -L/usr/lib/x86_64-linux-gnu
     -L/usr/lib/../lib
     -L/usr/lib/gcc/x86_64-linux-gnu/11/../../..
     test.o
     -lstdc++
     -lm
     -lgcc_s
     -lgcc
     -lc
     -lgcc_s
     -lgcc
     /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
     /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (5)

```
ld
      -plugin /usr/lib/gcc/x86_64-linux-gnu/11/liblto_plugin.so
      -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/11/lto-wrapper
      -plugin-opt=-fresolution=/tmp/cc0iAT08.res
      -plugin-opt=-pass-through=-lgcc_s
      -plugin-opt=-pass-through=-lgcc
      -plugin-opt=-pass-through=-lc
      -plugin-opt=-pass-through=-lgcc_s
      -plugin-opt=-pass-through=-lgcc
      --build-id
      --eh-frame-hdr
      -m elf_x86_64
      --hash-style=gnu
      --as-needed
      -dynamic-linker /lib64/ld-linux-x86-64.so.2
      -pie
      -z now
      -z relro
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crti.o
      /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
      -L/usr/lib/gcc/x86_64-linux-gnu/11
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../../lib
      -L/lib/x86_64-linux-gnu -L/lib/../lib
      -L/usr/lib/x86_64-linux-gnu
      -L/usr/lib/../lib
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../..
      test.o
      -lstdc++
      -lm
      -lgcc_s
      -lgcc
      -lc
      -lgcc_s
      -lgcc
      /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (6)

```
ld
     --build-id
     --eh-frame-hdr
     -m elf_x86_64
     --hash-style=gnu
     --as-needed
     -dynamic-linker /lib64/ld-linux-x86-64.so.2
     -pie
     -z now
     -z relro
     /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o
     /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crti.o
     /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
     -L/usr/lib/gcc/x86_64-linux-gnu/11
     -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu
     -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../../lib
     -L/lib/x86_64-linux-gnu -L/lib/../lib
     -L/usr/lib/x86_64-linux-gnu
     -L/usr/lib/../lib
     -L/usr/lib/gcc/x86_64-linux-gnu/11/../../..
     test.o
     -lstdc++
     -lm
     -lgcc_s
     -lgcc
     -lc
     -lgcc_s
     -lgcc
     /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
     /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (6)

```
ld
      --build-id
      --eh-frame-hdr
      -m elf_x86_64
      --hash-style=gnu
      --as-needed
      -dynamic-linker /lib64/ld-linux-x86-64.so.2
      -pie
      -z now
      -z relro
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crti.o
      /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
      -L/usr/lib/gcc/x86_64-linux-gnu/11
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../../lib
      -L/lib/x86_64-linux-gnu -L/lib/../lib
      -L/usr/lib/x86_64-linux-gnu
      -L/usr/lib/../lib
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../..
      test.o
      -lstdc++
      -lm
      -lgcc_s
      -lgcc
      -lc
      -lgcc_s
      -lgcc
      /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (7)

```
ld
      -dynamic-linker /lib64/ld-linux-x86-64.so.2
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crti.o
      /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
      -L/usr/lib/gcc/x86_64-linux-gnu/11
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../../lib
      -L/lib/x86_64-linux-gnu -L/lib/../lib
      -L/usr/lib/x86_64-linux-gnu
      -L/usr/lib/../lib
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../..
      test.o
      -lstdc++
      -lm
      -lgcc_s
      -lgcc
      -lc
      -lgcc_s
      -lgcc
      /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (8)

```
ld
      -dynamic-linker /lib64/ld-linux-x86-64.so.2
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crti.o
      /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
      -L/usr/lib/gcc/x86_64-linux-gnu/11
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../../lib
      -L/lib/x86_64-linux-gnu -L/lib/../lib
      -L/usr/lib/x86_64-linux-gnu
      -L/usr/lib/../lib
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../..
      test.o
      -lstdc++
      -lm
      -lgcc_s
      -lgcc
      -lc
      -lgcc_s
      -lgcc
      /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (9)

```
unsigned __int128 foo(unsigned __int128 a,
                      unsigned __int128 b)
{
    return a + b;
}

unsigned __int128 bar(unsigned __int128 a,
                      unsigned __int128 b)
{
    return a / b;
}
```

```
foo:
        mov     rax, rsi
        mov     r8, rdi
        mov     rdi, rax
        mov     rax, rdx
        mov     rdx, rcx
        add     rax, r8
        adc     rdx, rdi
        ret
bar:
        sub     rsp, 8
        call    __udivti3
        add     rsp, 8
        ret
```

Libraries (10)

```
ld
    -dynamic-linker /lib64/ld-linux-x86-64.so.2
    /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o
    /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crti.o
    /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
    -L/usr/lib/gcc/x86_64-linux-gnu/11
    -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu
    -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../../lib
    -L/lib/x86_64-linux-gnu -L/lib/../lib
    -L/usr/lib/x86_64-linux-gnu
    -L/usr/lib/../lib
    -L/usr/lib/gcc/x86_64-linux-gnu/11/../../..
    test.o
    -lc
    /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
    /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (11)

```
ld
      -dynamic-linker /lib64/ld-linux-x86-64.so.2
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crti.o
      /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
      -L/usr/lib/gcc/x86_64-linux-gnu/11
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../../../lib
      -L/lib/x86_64-linux-gnu -L/lib/../lib
      -L/usr/lib/x86_64-linux-gnu
      -L/usr/lib/../lib
      -L/usr/lib/gcc/x86_64-linux-gnu/11/../../..
      test.o
      -lc
      /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
      /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (12)

```
ld
     -dynamic-linker /lib64/ld-linux-x86-64.so.2
     /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/Scrt1.o
     /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crti.o
     /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
     test.o
     -lc
     /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
     /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crtn.o
```

Libraries (13)

```
ld
     -dynamic-linker /lib64/ld-linux-x86-64.so.2
     /usr/lib/x86_64-linux-gnu/Scrt1.o
     /usr/lib/x86_64-linux-gnu/crti.o
     /usr/lib/gcc/x86_64-linux-gnu/11/crtbeginS.o
     test.o
     -lc
     /usr/lib/gcc/x86_64-linux-gnu/11/crtendS.o
     /usr/lib/x86_64-linux-gnu/crtn.o
```

Libraries (14)

```
$ nm -C /lib/x86_64-linux-gnu/libc.a  | grep printf
...
printf.o:
0000000000000000 T _IO_printf
0000000000000000 T __printf
                 U __vfprintf_internal
0000000000000000 T printf
...
```

Libraries (15)

```
$ ldd ./a.out
        linux-vdso.so.1 (0x00007ffd25fc5000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f57907ac000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f57909e5000)
```

Header files (1)

- To use a function from a different translation unit we need to have its declaration in our translation unit.
- Many usages in various translation units means many similar declarations.
  - In case the signature need to be changed all these declarations have to be changed.
  - Can be have a declaration in one place (.h file) and #include it from every translation unit that need this declaration.
  - .h files are pieces of text that can be included into other files.
    - Commonly one header file corresponds to one .cpp file.

Header files (2)

```
// greetings.h
void hello();
void привет();
```

```
// greetings.cpp
void hello()
{
    ...
}

void привет()
{
    ...
}
```

Header files (3)

Never #include one .cpp file into another.

```
// a.cpp
#include "b.cpp"

int main()
{
    f(42);
}
```

```
// b.cpp
#include <stdio.h>

void f(int a)
{
    printf("Answer: %d\n", a);
}
```

```
$ g++ a.cpp b.cpp
```

Header files (4)

Never #include one .cpp file into another.

```
// a.cpp
#include "b.cpp"

int main()
{
    f(42);
}
```

```
// b.cpp
#include <stdio.h>

void f(int a)
{
    printf("Answer: %d\n", a);
}
```

```
$ g++ a.cpp b.cpp
/usr/bin/ld: /tmp/ccATqQQn.o: in function `f(int)':
b.cpp:(.text+0x0): multiple definition of `f(int)'; /tmp/ccfeGKLs.o:a.cpp:
(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Header files (5)

Never #include one .cpp file into another.

```
// a.cpp
#include "b.cpp"

int main()
{
    f(42);
}
```

```
// b.cpp
#include <stdio.h>

void f(int a)
{
    printf("Answer: %d\n", a);
}
```

```
$ nm -C a.o
0000000000000000 T f(int)
000000000000002b T main
                 U printf
```

```
$ nm -C b.o
0000000000000000 T f(int)
                 U printf
```

Header files (6)

```
// a.cpp
#include <stdio.h>

void f(int a)
{
    printf("Answer: %d\n", a);
}

int main()
{
    f(42);
}
```

```
// b.cpp
#include <stdio.h>

void f(int a)
{
    printf("Answer: %d\n", a);
}
```

```
$ nm -C a.o
0000000000000000 T f(int)
000000000000002b T main
                 U printf
```

```
$ nm -C b.o
0000000000000000 T f(int)
                 U printf
```

Header files (7)

Never #include one .cpp file into another.

- Inclusion of .cpp likely leads to redefinition errors.
- Only #include header files.

Header files (8)

Never write function defintions in header files.

```
// hello.h
void say_hello()
{
}
```

```
// a.cpp
#include "hello.h"

int main()
{
    say_hello();
}
```

```
// b.cpp
#include "hello.h"
```

```
$ g++ a.cpp b.cpp
```

Header files (9)

Never write function defintions in header files.

```
// hello.h
void say_hello()
{
}
```

```
// a.cpp
#include "hello.h"

int main()
{
    say_hello();
}
```

```
// b.cpp
#include "hello.h"
```

```
$ g++ a.cpp b.cpp
/usr/bin/ld: /tmp/ccCGIbHJ.o: in function `say_hello()':
b.cpp:(.text+0x0): multiple definition of `say_hello()';
/tmp/cccQ6TDZ.o:a.cpp:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Header files (10)

.cpp are the files which compiler translates.

.h are the files used in `#include` directives.

- Never `#include` .cpp-file into another .cpp-file.
  - This causes multiple definitions.
- Never write definitions in .h files.
  - This causes multiple definitions too.

Classes (1)

In C classes don't generate code on their own, but they can affect the code generated by functions.

```
struct point
{
    int32_t x;
    int32_t y;
};

void foo(point* p)
{
    p->y = 42;
}
```

```
foo(point*):
        mov     DWORD PTR [rdi+4], 42
        ret
```

```
struct point
{
    int64_t x;
    int64_t y;
};

void foo(point* p)
{
    p->y = 42;
}
```

```
foo(point*):
        mov     QWORD PTR [rdi+8], 42
        ret
```

Classes (2)

In C classes don’t generate code on their own, but they can affect the code generated by functions.

- Therefore classes are normally written in header files.

Terminology: for classes the terms declaration and definition are used interchangeably.

Include guards (1)

```
// x.h
struct x
{
  int a;
  int b;
};
```

```
// y.h
#include "x.h"

struct y
{
  x p, q;
};
```

```
// z.h
#include "x.h"

struct z
{
  x xx;
  int size;
};
```

```
// main.cpp
#include "y.h"
#include "z.h"

int main()
{}
```

Include guards (2)

```
// x.h
struct x
{
  int a;
  int b;
};
```

```
// y.h
#include "x.h"

struct y
{
  x p, q;
};
```

```
// z.h
#include "x.h"

struct z
{
  x xx;
  int size;
};
```

```
// main.cpp
#include "y.h"
#include "z.h"

int main()
{}
```

```
$ g++ main.cpp
In file included from z.h:2,
                 from main.cpp:3:
x.h:2:8: error: redefinition of
‘struct x’
    2 | struct x
      |        ^
In file included from y.h:2,
                 from main.cpp:2:
x.h:2:8: note: previous definition of
‘struct x’
    2 | struct x
      |        ^
```

Include guards (3)

```
// x.h
struct x
{
  int a;
  int b;
};
```

```
// y.h
#include "x.h"

struct y
{
  x p, q;
};
```

```
// z.h
#include "x.h"

struct z
{
  x xx;
  int size;
};
```

```
// main.cpp
#include "y.h"
#include "z.h"

int main()
{}
```

```
$ g++ -E -P main.cpp
struct x
{
  int a;
  int b;
};
struct y
{
  x p, q;
};
struct x
{
  int a;
  int b;
};
struct z
{
  x xx;
  int size;
};
int main()
{}
```

Include guards (4)

```
// x.h
#ifndef X_H
#define X_H

struct x
{
  int a;
  int b;
};

#endif
```

```
// y.h
#include "x.h"

struct y
{
  x p, q;
};
```

```
// z.h
#include "x.h"

struct z
{
  x xx;
  int size;
};
```

```
// main.cpp
#include "y.h"
#include "z.h"

int main()
{}
```

```
$ g++ -E -P main.cpp
struct x
{
  int a;
  int b;
};
struct y
{
  x p, q;
};
struct z
{
  x xx;
  int size;
};
int main()
{}
```

Include guards (5)

```
// x.h
#pragma once

struct x
{
  int a;
  int b;
};
```

```
// y.h
#include "x.h"

struct y
{
  x p, q;
};
```

```
// z.h
#include "x.h"

struct z
{
  x xx;
  int size;
};
```

```
$ g++ -E -P main.cpp
struct x
{
  int a;
  int b;
};
struct y
{
  x p, q;
};
struct z
{
  x xx;
  int size;
};
int main()
{}
```

```
// main.cpp
#include "y.h"
#include "z.h"

int main()
{}
```

Include guards (6)

- Designing headers to allow being included multiple times is considered good
- Standard conforming way of doing this is using include guards

```
#ifndef XXX_H
#define XXX_H
…
#endif
```

- #pragma once has the same effect, but is non-standard
  - It is supported by all major compilers though
- Do not write include guards (or #pragma once) in .cpp files

Recursive structs and #include loops (1)

```
// a.h
#ifndef A_H
#define A_H
#include "b.h"

struct a
{
    b* bb;
};

#endif
```

```
// b.h
#ifndef B_H
#define B_H
#include "a.h"

struct b
{
    a* aa;
};

#endif
```

```
// main.cpp
#include "a.h"
#include "b.h"


$ g++ main.cpp
```

Recursive structs and #include loops (2)

```
// a.h
#ifndef A_H
#define A_H
#include "b.h"

struct a
{
    b* bb;
};

#endif
```

```
// b.h
#ifndef B_H
#define B_H
#include "a.h"

struct b
{
    a* aa;
};

#endif
```

```
// main.cpp
#include "a.h"
#include "b.h"
```

```
$ g++ main.cpp
In file included from a.h:3,
                 from main.cpp:1:
b.h:7:5: error: ‘a’ does not name a type
    7 |     a* aa;
      |     ^
```

Recursive structs and #include loops (3)

```
// a.h
#ifndef A_H
#define A_H
#include "b.h"

struct a
{
    b* bb;
};

#endif
```

```
// b.h
#ifndef B_H
#define B_H
#include "a.h"

struct b
{
    a* aa;
};

#endif
```

```
// main.cpp
#include "a.h"
#include "b.h"
```

```
$ g++ -E -P main.cpp
struct b
{
    a* aa;
};
struct a
{
    b* bb;
};
```

Recursive structs and #include loops (4)

```
// a.h
#ifndef A_H
#define A_H
struct b;

struct a
{
    b* bb;
};

#endif
```

```
// b.h
#ifndef B_H
#define B_H
struct a;

struct b
{
    a* aa;
};

#endif
```

```
// main.cpp
#include "a.h"
#include "b.h"
```

```
$ g++ -E -P main.cpp
struct b;
struct a
{
    b* bb;
};
struct a;
struct b
{
    a* aa;
};
```

Recursive structs and #include loops (5)

The declarations of form `struct x;` are called forward declarations.

They declare the class without specifying its members.

Recursive structs and #include loops (6)

```
// Enough forward declaration

struct x;

x* p;
x foo();
void bar(x);
using y = x;
```

```
// Require regular declaration

struct x
{
    int a;
};

x q;
x foo() {}
void bar(x) {}
p->a;
```

Recursive structs and #include loops (7)

- A loop of #include directives is likely a bug.
  - Whenever you get such a loop consider using forward declarations.
- Some projects encourage using forward declaration instead of #include's whenever possible.
  - #include's inside other headers are the worst offenders increasing the amount of text that compiler have to process.
    - More text to process means longer compilation time.
    - More dependent files means more files to recompile when the header is changed.

Struct mismatch (1)

- All declarations of the class with the same name must be equivalent in different translation units.
  - Compiler can not diagnose this during translation.
  - The standard doesn’t require a diagnostic.
  - This is a part of ODR (one defintion rule).

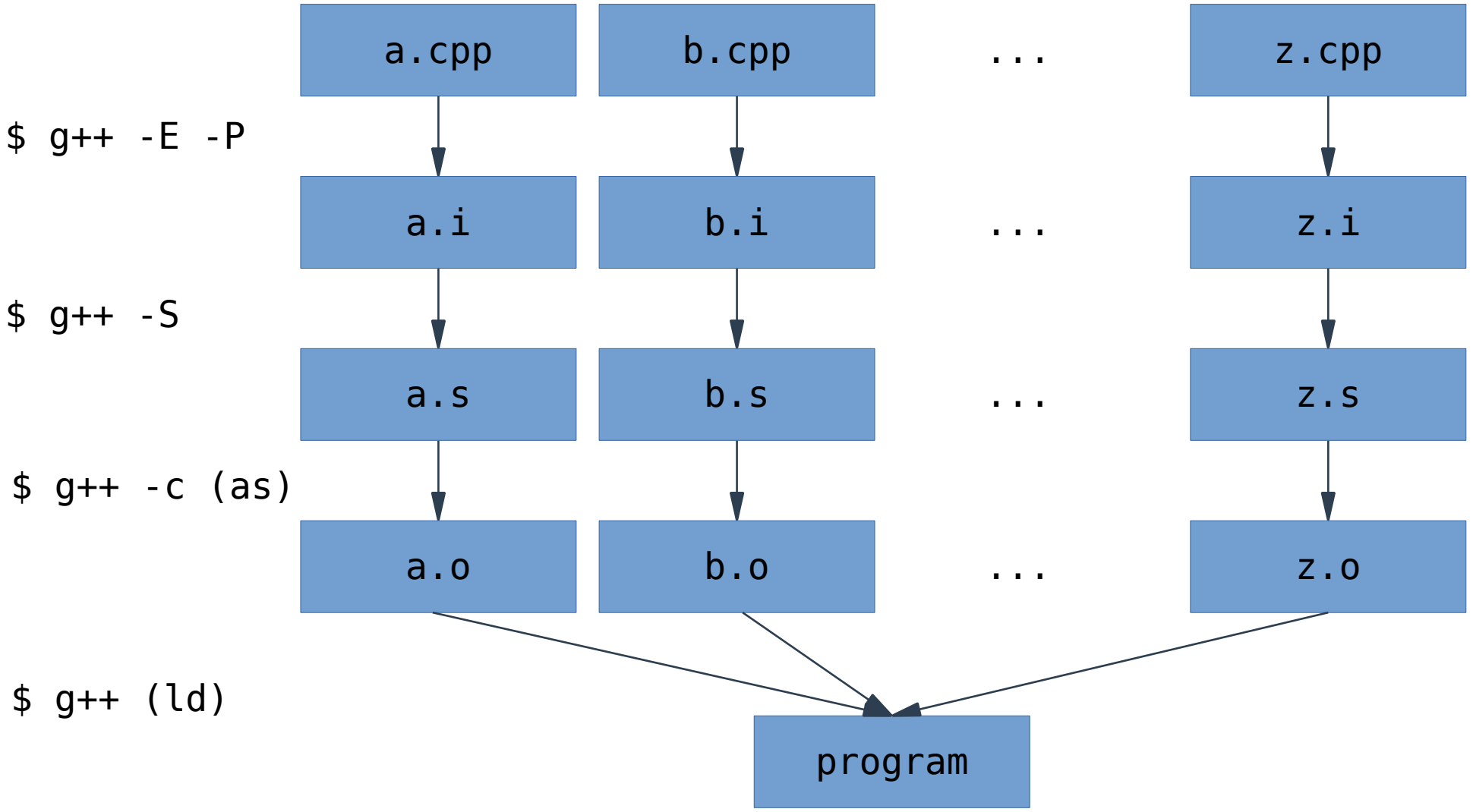
Struct mismatch (2)

- One definition rule:
  - No translation unit shall contain more than one definition of any variable, function, class type.
  - Every program shall contain exactly one definition of every function or variable.
  - There can be more than one definition of a class type, but they all must be equivalent.

Struct mismatch (3)

- One definition rule:
  - No translation unit shall contain more than one definition of any variable, function, class type.
  - Every program shall contain exactly one definition of every function or variable.
  - There can be more than one definition of a class type.
    - each definition of D shall consist of the same sequence of tokens; and
    - in each definition of D, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of D, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a const object with internal or no linkage if the object has the same literal type in all definitions of D, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of D; and
    - in each definition of D, corresponding entities shall have the same language linkage; and
    - in each definition of D, the overloaded operators referred to, the implicit calls to conversion functions, constructors, operator new functions and operator delete functions, shall refer to the same function, or to a function defined within the definition of D; and
    - in each definition of D, a default argument used by an (implicit or explicit) function call is treated as if its token sequence were present in the definition of D; that is, the default argument is subject to the three requirements described above (and, if the default argument has sub-expressions with default arguments, this requirement applies recursively).
    - if D is a class with an implicitly-declared constructor (12.1), it is as if the constructor was implicitly defined in every translation unit where it is odr-used, and the implicit definition in every translation unit shall call the same constructor for a base class or a class member of D.

Struct mismatch (4)



Struct mismatch (5)

```
// a.cpp
#include <iostream>

struct x
{
  int a;
  double b;
  int c;
  int d;
};

x f();

int main()
{
  x xx = f();
  std::cout << xx.a << " "
            << xx.b << " "
            << xx.c << " "
            << xx.d << "\n";
}
```

```
// b.cpp
struct x
{
  int a;
  int b;
  int c;
  int d;
  int e;
};

x f()
{
  x result;
  result.a = 1;
  result.b = 2;
  result.c = 3;
  result.d = 4;
  result.e = 5;
  return result;
}
```

```
$ ./a.out
```

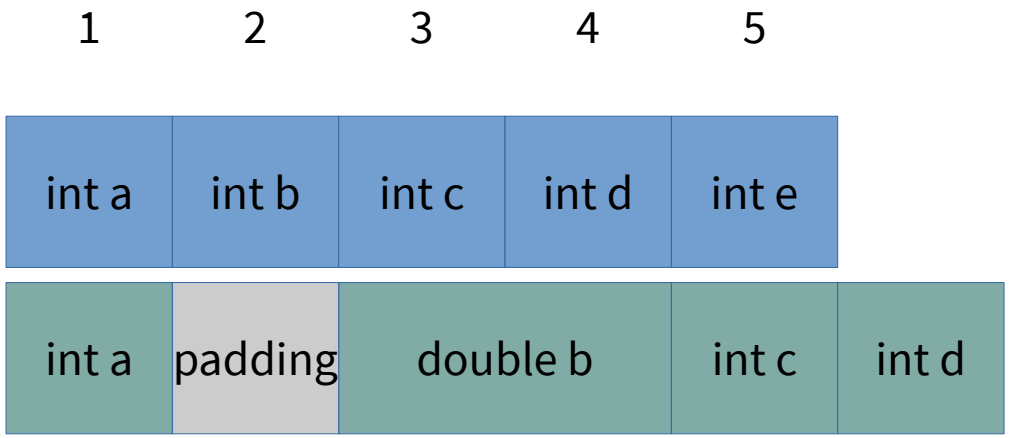
Struct mismatch (6)

```
// a.cpp
#include <iostream>

struct x
{
  int a;
  double b;
  int c;
  int d;
};

x f();

int main()
{
  x xx = f();
  std::cout << xx.a << " "
            << xx.b << " "
            << xx.c << " "
            << xx.d << "\n";
}
```

```
// b.cpp
struct x
{
  int a;
  int b;
  int c;
  int d;
  int e;
};

x f()
{
  x result;
  result.a = 1;
  result.b = 2;
  result.c = 3;
  result.d = 4;
  result.e = 5;
  return result;
}
```

```
$ ./a.out
1 8.48798e-314 5 0
```

Struct mismatch (7)



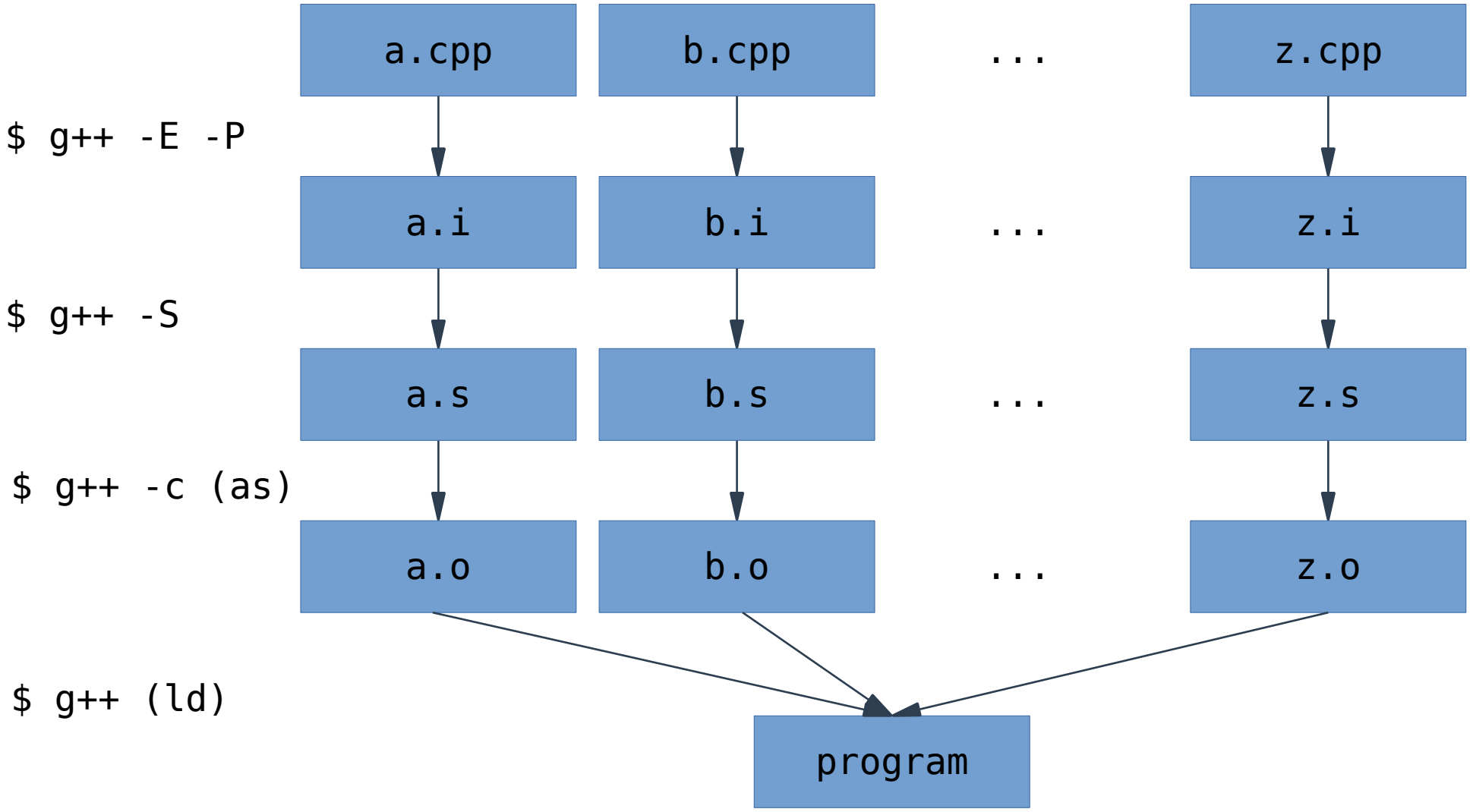

```
$ ./a.out
1 8.48798e-314 5 0
```

Inlining (1)

```
#include <cstdint>

int32_t sum(int32_t a, int32_t b)
{
    return a + b;
}

int32_t test(int32_t a, int32_t b)
{
    return sum(a - b, b);
}
```

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

test(int, int):
        mov     eax, edi
        ret
```

Inlining (2)



Inlining (3)

- Traditionally inlining didn't work across translation units:
  - At the point of usage the body of the function is unknown.
  - At the point of definition the usages are unknown.
- Nowadays we have LTO (link time optimization)
  - The compiler defers code generation to the point of linking.
    - The compiler writes to the object files an IR (intemediate representation) instead of machine code.
    - The linker reads the IR for all object files, does optimization across translation units and writes the final binary.
  - With LTO compiler can detect some types of ODR-violations.
  - LTO usually negatively affects recompilation time

Inlining (4)

```
// a.cpp
#include <iostream>

struct x
{
  int a;
  double b;
  int c;
  int d;
};

x f();

int main()
{
  x xx = f();
  std::cout << xx.a << " "
            << xx.b << " "
            << xx.c << " "
            << xx.d << "\n";
}
```

```
// b.cpp
struct x
{
  int a;
  int b;
  int c;
  int d;
  int e;
};

x f()
{
  x result;
  result.a = 1;
  result.b = 2;
  result.c = 3;
  result.d = 4;
  result.e = 5;
  return result;
}
```

Inlining (5)

```
$ g++ -flto a.cpp b.cpp
a.cpp:4:8: warning: type ‘struct x’ violates the C++ One Definition Rule [-Wodr]
    4 | struct x
      |        ^
b.cpp:2:8: note: a different type is defined in another translation unit
    2 | struct x
      |        ^
a.cpp:7:10: note: the first difference of corresponding definitions is field ‘b’
    7 |   double b;
      |          ^
b.cpp:5:7: note: a field of same name but different type is defined in another translation unit
    5 |   int b;
      |       ^
a.cpp:4:8: note: type ‘double’ should match type ‘int’
    4 | struct x
      |        ^
a.cpp:12:3: warning: ‘f’ violates the C++ One Definition Rule [-Wodr]
   12 | x f();
      |   ^
b.cpp:11:3: note: return value type mismatch
   11 | x f()
      |   ^
b.cpp:2:8: note: type ‘struct x’ itself violates the C++ One Definition Rule
    2 | struct x
      |        ^
a.cpp:4:8: note: the incompatible type is defined here
    4 | struct x
      |        ^
b.cpp:11:3: note: ‘f’ was previously declared here
   11 | x f()
      |   ^
b.cpp:11:3: note: code may be misoptimized unless ‘-fno-strict-aliasing’ is used
```

Inlining (6)

- Traditionally inlining didn’t work across translation units:
  - At the point of usage the body of the function is unknown.
  - At the point of definition the usages are unknown.
- Let us assume that the LTO haven’t been invented yet and we want to inline some functions.
- Can we have a definition in header file?
  - Yes, but it will cause redefinition errors.
  - The declaration specifier `inline` supresses these errors.

Inlining (7)

```
// f.h
#pragma once
#include <cstdio>

inline void f()
{
    printf("Hello, world!\n");
}
```

```
$ g++ a.cpp b.cpp
$ ./a.out
Hello, world!
Hello, world!
```

```
//a.cpp
#include "f.h"

void g();

int main()
{
    f();
    g();
}
```

```
// b.cpp
#include "f.h"

void g()
{
    f();
}
```

Inlining (8)

- One definition rule:
  - No translation unit shall contain more than one definition of any variable, function, class type.
  - Every program shall contain exactly one definition of every non-inline function or variable.
  - There can be more than one definition of a class type or inline function or variable, but they all must be equivalent.

Inlining (9)

```
// a.cpp
#include <cstdio>

inline void f()
{
    printf("Hello, a.cpp!\n");
}

void g();

int main()
{
    f();
    g();
}
```

```
// b.cpp
#include <cstdio>

inline void f()
{
    printf("Hello, b.cpp!\n");
}

void g()
{
    f();
}
```

```
$ g++ a.cpp b.cpp
$ ./a.out
Hello, a.cpp!
Hello, a.cpp!
```

```
$ g++ b.cpp a.cpp
$ ./a.out
Hello, b.cpp!
Hello, b.cpp!
```

```
$ g++ -O2 a.cpp b.cpp
$ ./a.out
Hello, a.cpp!
Hello, b.cpp!
```

Inlining (10)

- Since C++17 inline variables exists too.
- Their semantics is exactly the same as for inline functions.
  - Definitions in different translation units are allowed, but must be equivalent.

Best practices (1)

- Don’t forget include guards (or #pragma once) in headers.
  - Don’t use include guards (or #pragma once) in .cpp files.
- Prefer writing declarations in headers and definitions in .cpp files.
  - If writing a definition in header file is needed use `inline`.
  - Exception: the definition of template functions is written in headers (and they are automatically `inline`).
    - Exception-exception: the definition of template functions can be written in .cpp file if used only in this file
- Never write static variables or functions in headers.
- Never #include .cpp files.
- If the function or variable is local to the .cpp file make it static (or put it into an anonymous namespace)

Best practices (2)

- #include <> and #include "" are two different forms of #include.
- They differs by the directories in which the header is searched.
  - #include <> searches in some include paths.
  - #include "" first searches in the currect directory, then in some other paths (usually empty set) and then in the same paths as for #include <>.
- Commonly
  - #include <> is used for headers from other libraries.
  - #include "" is used for headers in this current library.

C++20 modules

```
// hello.ixx
export module hello;
import std;

export void say_hello()
{
    std::cout << "Hello, world!\n";
}
```

```
// main.cpp
import hello;

int main()
{
    say_hello();
}
```

- C++20 modules don’t requre all this discipline required for header files.
  - Supported by MSVC
  - Partial support from GCC and clang.

Global variables

- Global variables might require dynamic initialization.
  - Their constructors are called before entering main()
  - Their destructors are called after exiting main()
  - Exception in their constructor causes program termination before entering main()
  - Excessive use of dynamically initialized global variables can cause long program start-up time.
- In one translation units they are initialized from top to bottom and deinitialized in the opposite order.
- Their initialization order across different translation units is unspecified.
  - Excessive use of global variables that access each one in their constructors/destructors can cause problems. This is called SIOF (static initialization order fiasco).
  - Prefer using automatic variables and just pass pointers to them.