

Programming and Laboratory - 2

2. Tools

Giulio Ermanno Pibiri — giulioermanno.pibiri@unive.it

Nicola Prezza — nicola.prezza@unive.it

Department of Environmental Sciences, Informatics and Statistics

Overview

- C++ project directory layout
- Makefiles
- Setting up a project step-by-step
- Debugging — Part 1: asserts and Valgrind
- Debugging — Part 2: compiler warnings and sanitizers

C++ project directory layout

- A C++ project is a collection of header + source files.
- Header files:
 - define class interfaces / function signatures
 - hpp extension
 - put into `include/`
- Source files:
 - contain the implementation of the signatures defined in the corresponding `.hpp` files
 - cpp extension
 - put into `src/`

```
parent_project_folder/  
|  
|— include/  
|  
|— src/  
|  
|— build/  
|  
|— tools/
```

Other sub-directories are `test` and `external`, for example.

See here for a more detailed overview: <https://api.csswg.org/bikeshed/?force=1&url=https://raw.githubusercontent.com/vector-of-bool/pitchfork/develop/data/spec.bs#tld>.

Header and source files

- Header files (extension `.hpp`) are used to declare just the prototypes (also known as, “signatures” or “declarations”) of the functions whose body is defined elsewhere (in a `.cpp` file).

For example, this is a function’s prototype defined in the header file `"include/myfunction.hpp"`:

```
int myfunction(int x, double y);
```

- Header files are then included in the `cpp` files that need to use those functions (because the compiler requires that at least the prototype of a function is defined in the `cpp` file using that function).

The file `"src/myfunction.cpp"` will begin with

```
#include "myfunction.hpp"
```

This is exactly like writing

```
int myfunction(int x, double y);
```

directly in the `.cpp` file.

- The object files (`.o`) defining the function’s body have to be **linked** at compile time.

Step by step using the terminal

1. `mkdir myfunction-project; cd myfunction-project`
2. `mkdir include src tools build`
3. `g++ -std=c++11 -c src/myfunction.cpp -I include -o build/myfunction.o`
4. `g++ -std=c++11 build/myfunction.o tools/main.cpp -I include -o build/main`
5. `./build/main`

Step by step using the terminal

1. `mkdir myfunction-project; cd myfunction-project`

2. `mkdir include src tools build`

Q. What if we forget this?

3. `g++ -std=c++11 -c src/myfunction.cpp -I include -o build/myfunction.o`

4. `g++ -std=c++11 build/myfunction.o tools/main.cpp -I include -o build/main`

5. `./build/main`

Step by step using the terminal

1. `mkdir myfunction-project; cd myfunction-project`

2. `mkdir include src tools build`

Q. What if we forget this?

Q. What if we forget this?

3. `g++ -std=c++11 -c src/myfunction.cpp -I include -o build/myfunction.o`

4. `g++ -std=c++11 build/myfunction.o tools/main.cpp -I include -o build/main`

5. `./build/main`

Step by step using the terminal

1. `mkdir myfunction-project; cd myfunction-project`

2. `mkdir include src tools build`

Q. What if we forget this?

Q. What if we forget this?




3. `g++ -std=c++11 -c src/myfunction.cpp -I include -o build/myfunction.o`

4. `g++ -std=c++11 build/myfunction.o tools/main.cpp -I include -o build/main`

5. `./build/main`

Q. What if we forget this instead?

Step by step using the terminal

1. `mkdir myfunction-project; cd myfunction-project`
2. `mkdir include src tools build`

3. `g++ -std=c++11 -c src/myfunction.cpp -I include -o build/myfunction.o`

4. `g++ -std=c++11 build/myfunction.o tools/main.cpp -I include -o build/main`

5. `./build/main`

We can replace step 3. and 4. with a single command

```
g++ -std=c++11 src/myfunction.cpp tools/main.cpp -I include -o build/main
```

that does **compilation and linking together in one step.**

Note: the object files can be deleted after linking since the linker links all of them together into the executable.

File by file compilation vs. compilation+linking in one step

1. Compiling file by file and linking them together afterwards:
 - only specific files have to be recompiled when needed instead of all of them each time (reduced compilation costs).
 - think about a real-world project with dozens of files...
2. Compilation+linking:
 - one command only, but
 - everything is recompiled each time and re-linked together.

How to scale up to larger projects?

- Using the methodology “*step by step using the terminal*” is great for learning but very **slow** for large projects:
- typing by hand every compilation command in the terminal is tedious and **error-prone**.
- To automate the compilation+linking pipeline, we can use a tool known as “Make”.

Make

<https://www.gnu.org/software/make/manual/make.html>

- The program make can be used to **automate a pipeline** of tasks, each of which may depend on the output of other tasks.
- Example: a task t_1 produces file f_1 . Task t_2 requires f_1 and produces file f_2 . Hence, task t_2 cannot be run if file f_1 has not been created (i.e. if task t_1 is not run first)!
- Typically used to **compile code** (many intermediate object files).
- The pipeline of tasks is specified in a “special” file, called **Makefile**.

Makefiles

- A Makefile is a textual file composed by *rules*.
- A rule is made up of:
 - **target**: name of a file we want to generate *or* a more generic name for the rule (e.g., `clean`, `debug`, `release`, etc.)
 - **prerequisites**: file(s) needed as input to generate the target or name(s) of rules needed to be executed before
 - **recipe**: an action to be performed (e.g., one or more compilation commands)
- File structure:

`<target>: <prerequisites>`

`<TAB><recipe>`



A TAB character (“\t”) is required here!

Let's add a Makefile to our example project

Here is the basic file:

```
1  build/myfunction.o: src/myfunction.cpp include/myfunction.hpp
2  ——— g++ -std=c++11 -c src/myfunction.cpp -I include -o build/myfunction.o
3
4  build/mytool: build/myfunction.o tools/main.cpp
5  ——— g++ -std=c++11 build/myfunction.o tools/main.cpp -I include -o build/mytool
6  |
```

Let's add a Makefile to our example project

Here is the basic file:

```
1  build/myfunction.o: src/myfunction.cpp include/myfunction.hpp
2  ——— g++ -std=c++11 -c src/myfunction.cpp -I include -o build/myfunction.o
3
4  build/mytool: build/myfunction.o tools/main.cpp
5  ——— g++ -std=c++11 build/myfunction.o tools/main.cpp -I include -o build/mytool
6  |
```

Q. What happens now if you type make from the project's parent directory?

Let's add a Makefile to our example project

Here is the basic file:

```
1  build/myfunction.o: src/myfunction.cpp include/myfunction.hpp
2  |—— g++ -std=c++11 -c src/myfunction.cpp -I include -o build/myfunction.o
3
4  build/mytool: build/myfunction.o tools/main.cpp
5  |—— g++ -std=c++11 build/myfunction.o tools/main.cpp -I include -o build/mytool
6  |
```

Q. What happens now if you type make from the project's parent directory?

A. Only the first rule (build/myfunction.o) is executed.
We can then type make build/mytool to execute the second.
Or, even better, add a new first rule all: build/mytool.

Let's add a Makefile to our example project

A refined version:

```
1  all: build/mytool
2
3  build/myfunction.o: src/myfunction.cpp include/myfunction.hpp
4  |-----g++ -std=c++11 -c src/myfunction.cpp -I include -o build/myfunction.o
5
6  build/mytool: build/myfunction.o tools/main.cpp
7  |-----g++ -std=c++11 build/myfunction.o tools/main.cpp -I include -o build/mytool
8  |
```

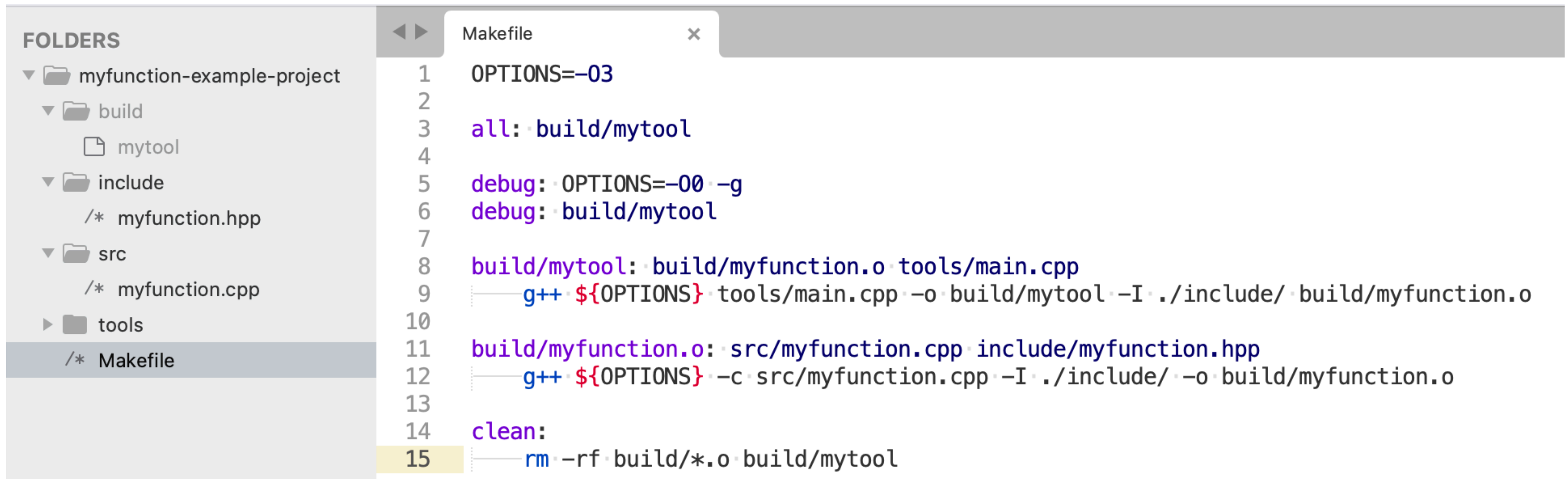
Q. What does this do instead?

So, in general...

- Behaviour of make <target>:
 1. If any prerequisite *changed* (timestamps are used) execute *recursively* the recipe of the prerequisite.
 2. Execute the recipe of the target.
- If no target specified (you just typed make): take the first one (and only the first one).

So, in general...

- Behaviour of make <target>:
 1. If any prerequisite *changed* (timestamps are used) execute *recursively* the recipe of the prerequisite.
 2. Execute the recipe of the target.
- If no target specified (you just typed make): take the first one (and only the first one).
- A complete example:



The screenshot shows a code editor with a project folder structure on the left and a Makefile in the center.

FOLDERS

- myfunction-example-project
 - build
 - mytool
 - include
 - /* myfunction.hpp
 - src
 - /* myfunction.cpp
 - tools
- /* Makefile

Makefile

```
1  OPTIONS=-O3
2
3  all: build/mytool
4
5  debug: OPTIONS=-O0 -g
6  debug: build/mytool
7
8  build/mytool: build/myfunction.o tools/main.cpp
9  |-----g++ ${OPTIONS} tools/main.cpp -o build/mytool -I ./include/ build/myfunction.o
10
11 build/myfunction.o: src/myfunction.cpp include/myfunction.hpp
12 |-----g++ ${OPTIONS} -c src/myfunction.cpp -I ./include/ -o build/myfunction.o
13
14 clean:
15 |-----rm -rf build/*.o build/mytool
```

Makefiles

Not just for compiling code!

- **Q.** What does the following Makefile do?

```
1  final.txt: text1.txt text2.txt
2  |—— cat text1.txt text2.txt > final.txt
3
4  text1.txt:
5  |—— echo "text 1" > text1.txt
6
7  text2.txt:
8  |—— echo "text 2" > text2.txt
9  |
```

Makefiles

Not just for compiling code!

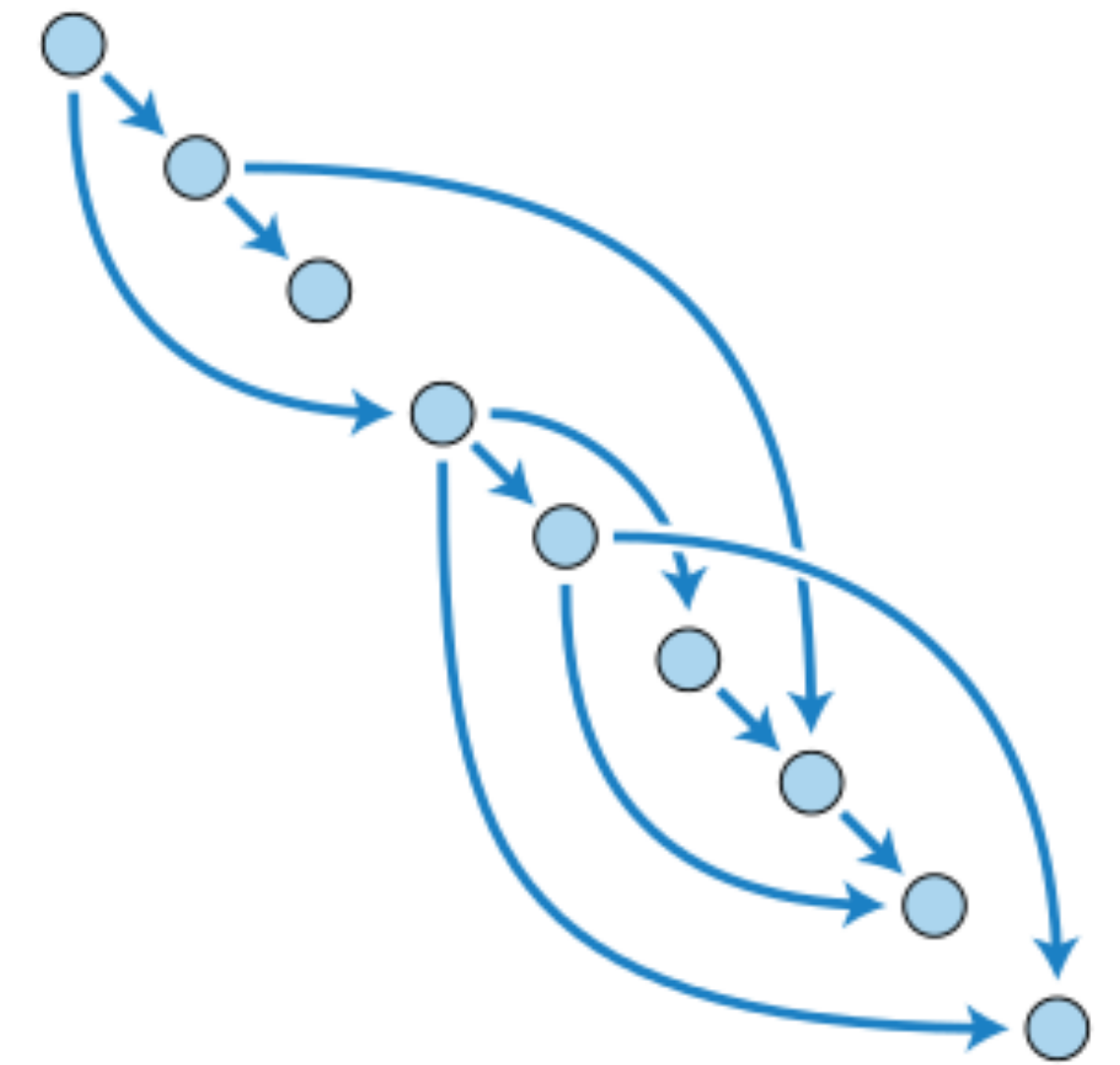
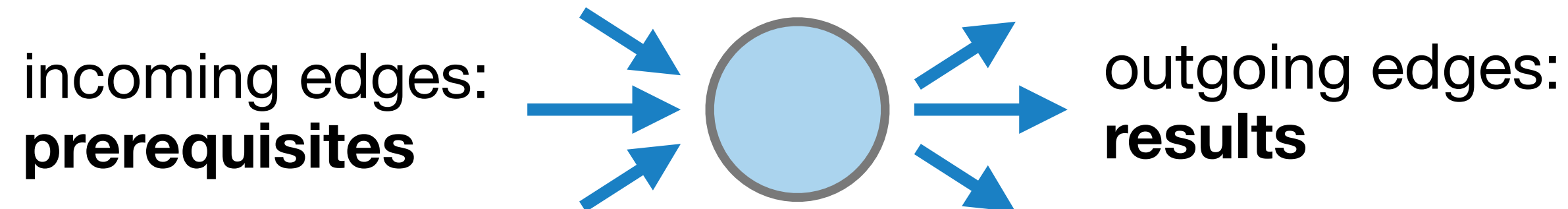
- **Q.** What does the following Makefile do?

```
1  final.txt: text1.txt text2.txt
2  ——— cat text1.txt text2.txt > final.txt
3
4  text1.txt:
5  ——— echo "text 1" > text1.txt
6
7  text2.txt:
8  ——— echo "text 2" > text2.txt
9  |
```

- **Q.** What happens if I run make two times?
- **Q.** What happens if I delete text1.txt and then run make?
- **Q.** What if I just touch (changes timestamp – check with `ls --full-time`) the file text1.txt?

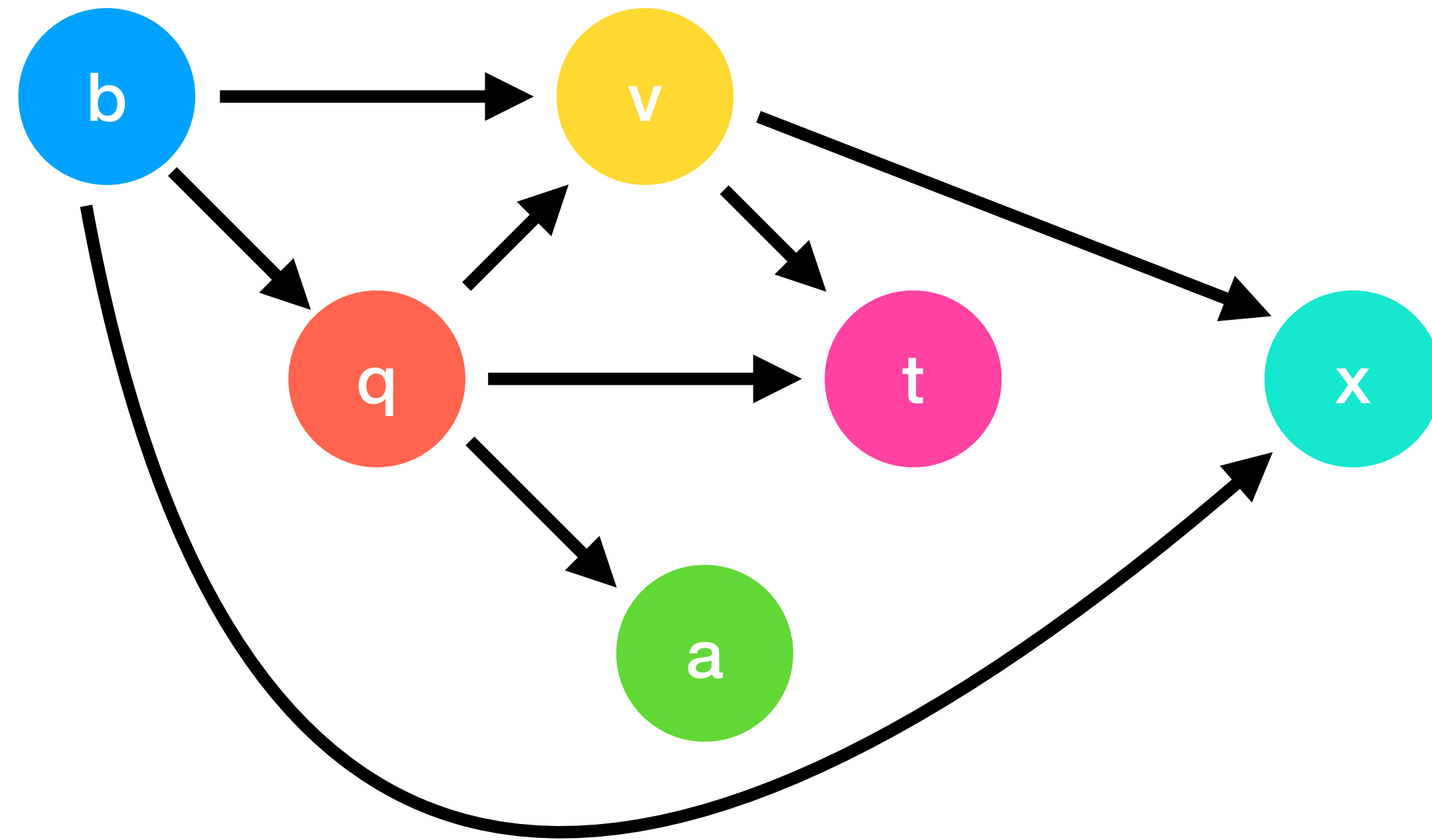
Makefiles as directed acyclic graphs (DAGs)

- The dependencies in Makefiles can be specified as **Directed Acyclic Graphs** (or DAGs).
- Basic model:



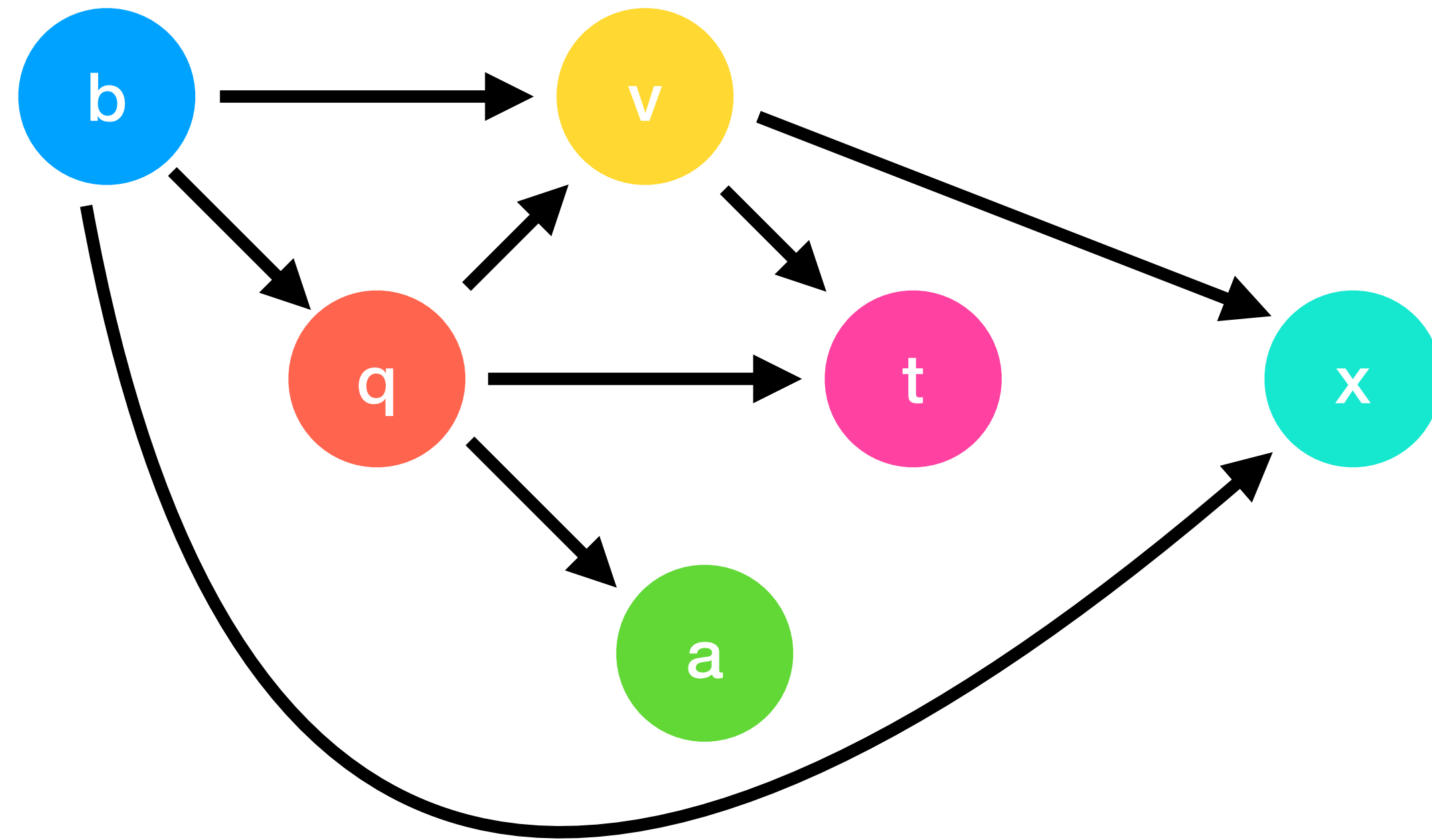
- To determine the order of execution of the targets, make sorts this graph so that arrows (dependencies) only go in one direction.
- This is called **topological sorting** (more details in CT0371) and is always possible on **acyclic** graphs.

DAG example

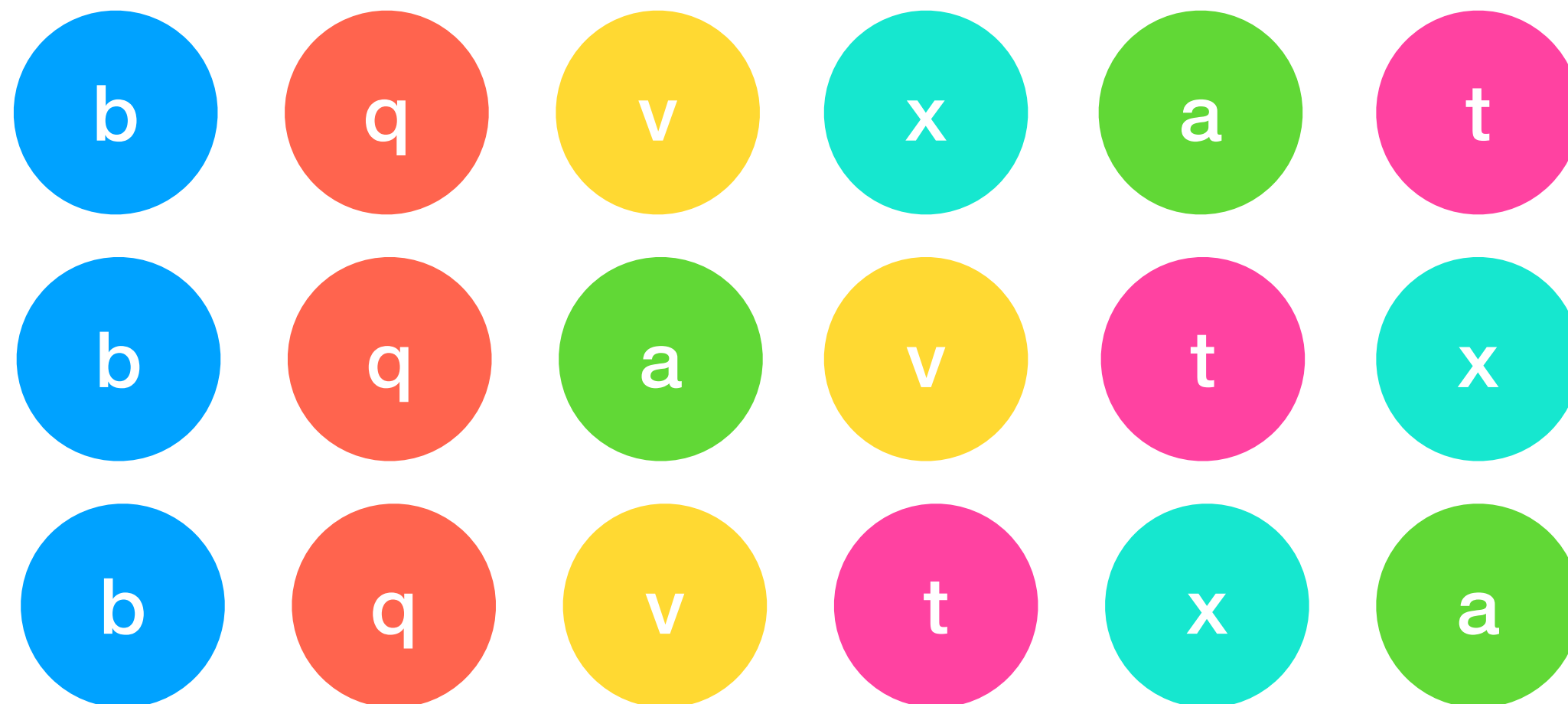


- Possible orders?

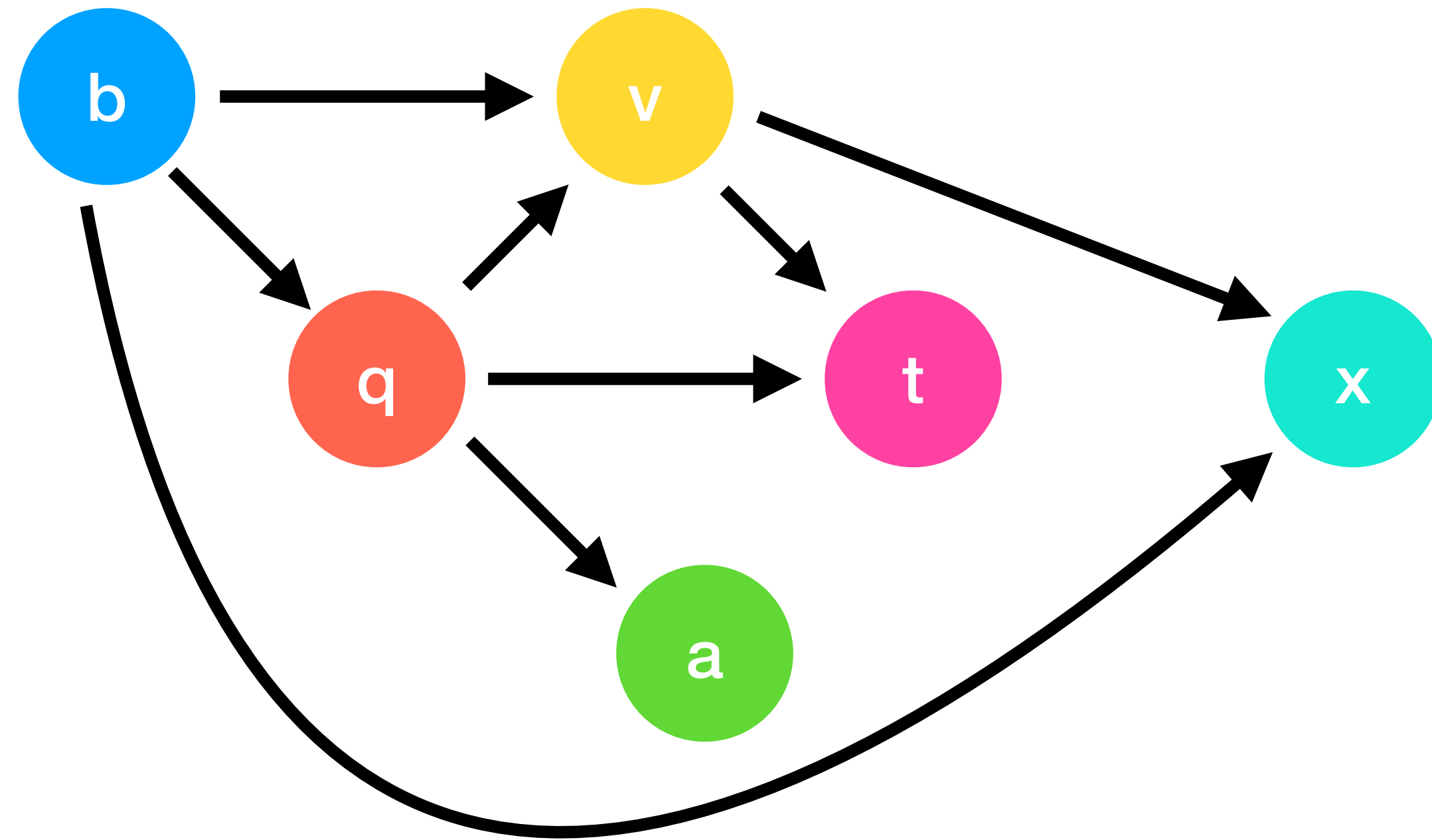
DAG example



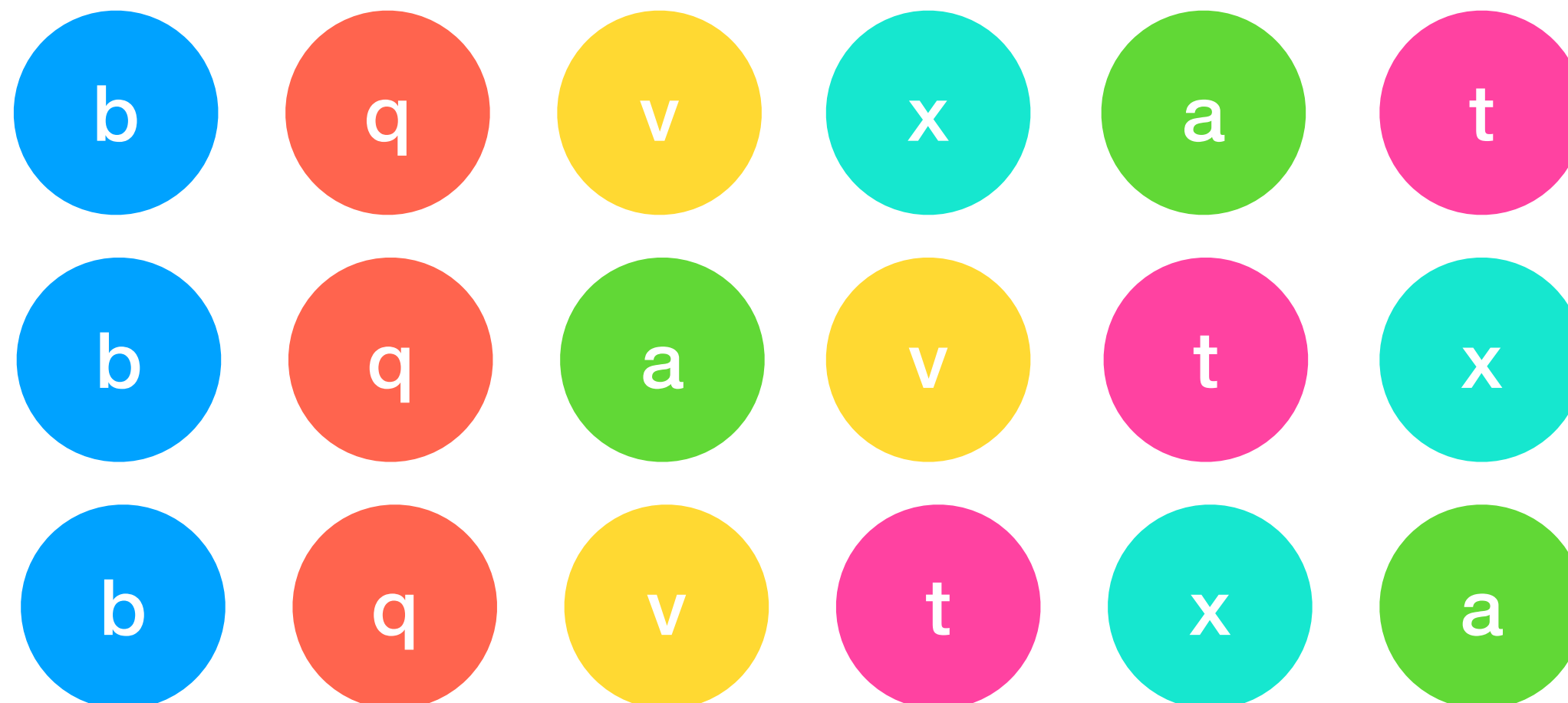
- Possible orders?



DAG example



- Possible orders?



Q. Can you describe an algorithm to output one of these orders?

Exercise

Consider the following Makefile and draw the corresponding DAG.

```
project: main.o utils.o command.o
    cc -o project main.o utils.o command.o
main.o: main.c defs.h
    cc -c main.c
utils.o: utils.c defs.h command.h
    cc -c utils.c
command.o: command.c defs.h command.h
    cc -c command.c
```

Setting-up a project step-by-step

Example step-by-step

1. In your home folder, create a folder `power-root-example-project/` and enter it.
2. Create the project directory layout (sub-directories: `include/`, `src/`, `tools/`, `build/`).
3. Create a header `include/myfunctions.hpp` declaring the prototypes of these functions:

```
int power(int, int);  
int root(int, int);
```

4. Create a source code file `src/power.cpp` defining the body of `int power(int x, int y)`, which returns x^y (use a for loop).
5. Create a source code file `src/root.cpp` defining the body of `int root(int x, int y)` which returns the largest integer z such that $z^y \leq x$ (use function `power` and a while loop).
6. Create a source code file `tools/main.cpp` containing the main function which computes the 3-rd root of 74088. The result has to be printed on the screen as "The answer is" followed by the result of `root(74088,3)`.
7. Create a Makefile that compiles everything in an executable called `theanswer`:
 - Specify the path of the includes using option `-I`.
 - Use optimization level 3.
 - Create a separate object file and a separate Makefile's target for each `.cpp` file.
 - Add also a `clean` target that removes all object files.
 - Optional: add other targets (e.g., `debug`) to compile with different options.
8. Compile and run the project.

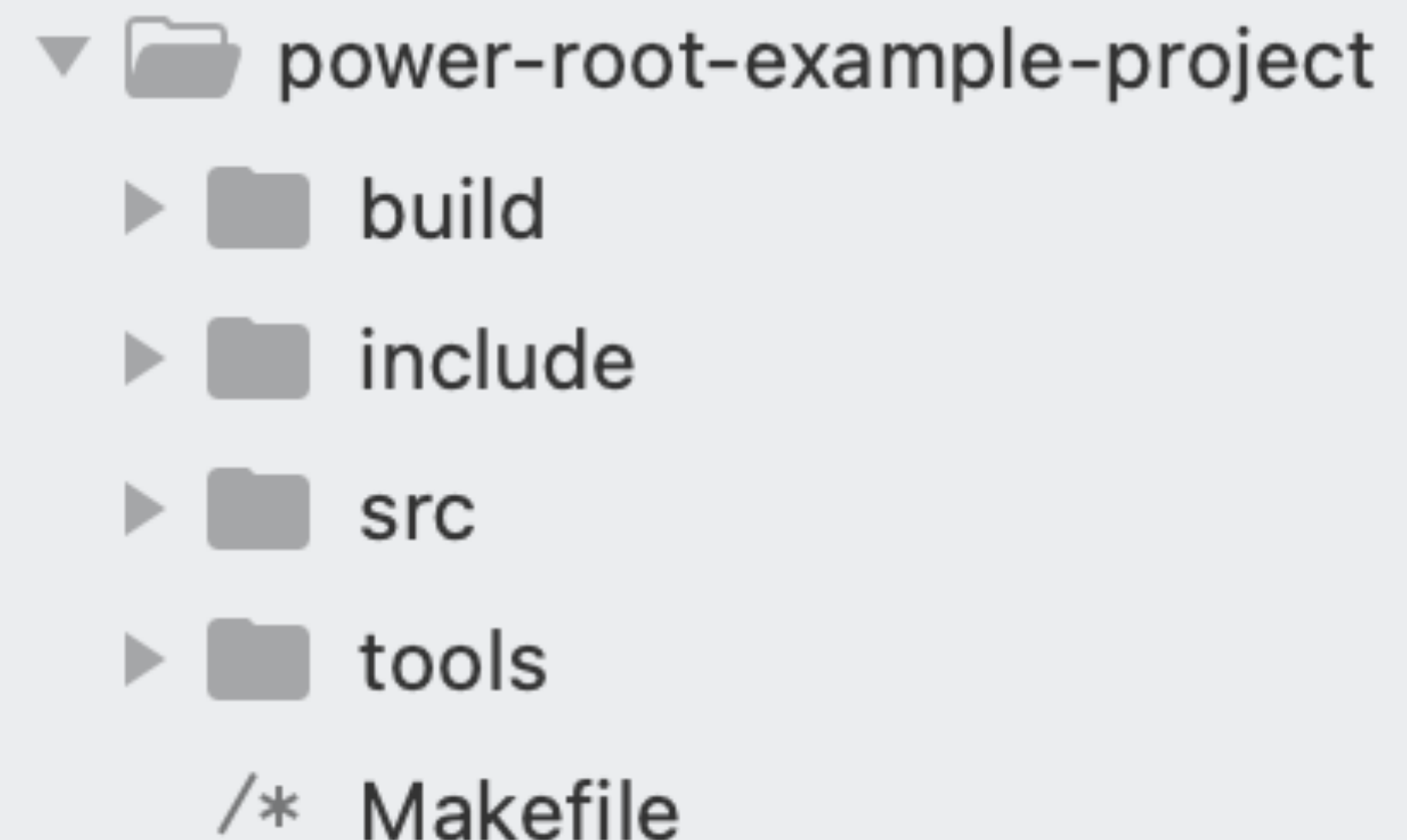
Step 1 and 2

In your home folder, create a folder `power-root-example-project/` and enter it.

Create the project directory layout (sub-directories: `include/`, `src/`, `tools/`, `build/`).

```
$ mkdir power-root-example-project; cd power-root-example-project  
$ mkdir include src tools build  
$ touch Makefile
```

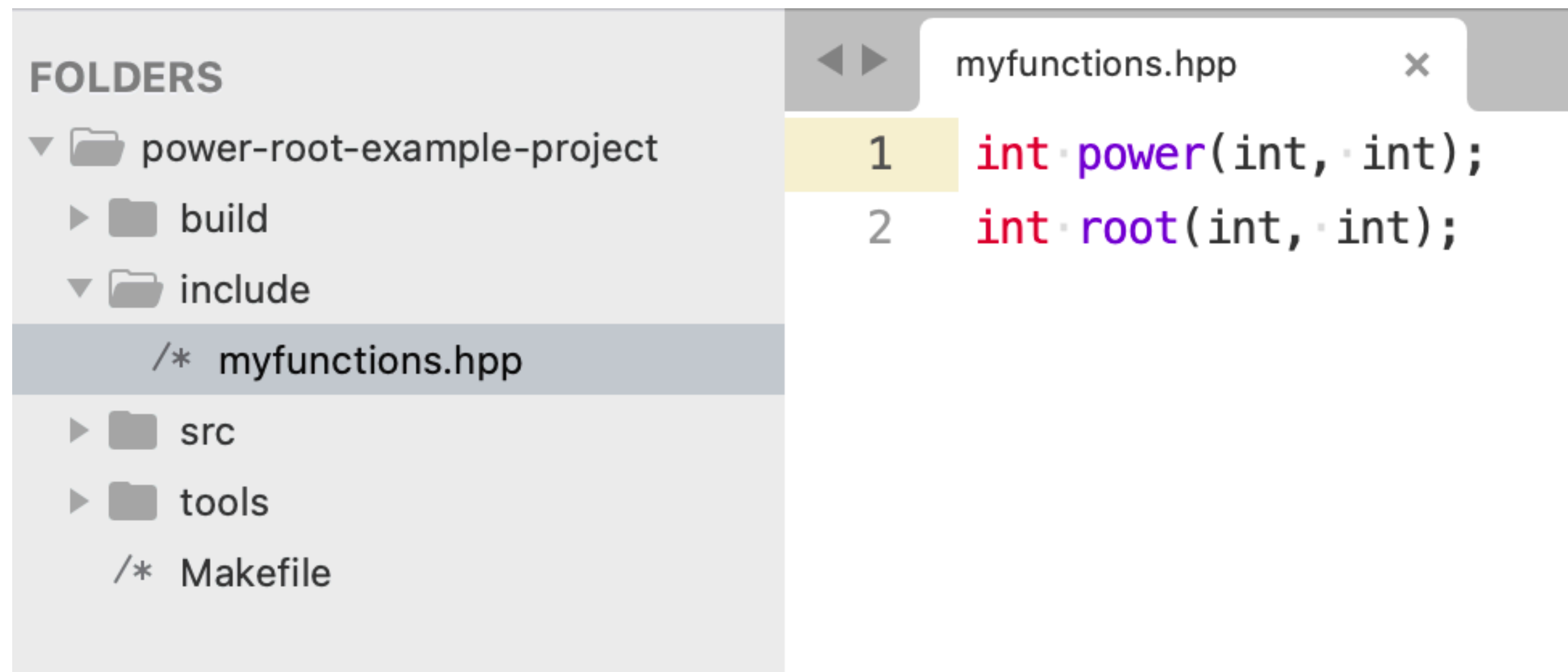
FOLDERS



Step 3

Create a header `include/myfunctions.hpp` declaring the prototypes of these functions:

```
int power(int, int);  
int root(int, int);
```



Step 4 and 5

Create a source code file `src/power.cpp` defining the body of `int power(int x, int y)`, which returns x^y (just use a for loop).

Create a source code file `src/root.cpp` defining the body of `int root(int x, int y)` which returns the largest integer z such that $z^y \leq x$ (use function `power` and a while loop).

The screenshot shows a code editor with two tabs: `power.cpp` and `root.cpp`. On the left, a 'FOLDERS' panel shows the project structure: `power-root-example-project` (expanded) containing `build`, `include` (containing `myfunctions.hpp`), `src` (containing `power.cpp` and `root.cpp`), and `tools` (containing `Makefile`). The `power.cpp` tab is active, showing the following code:

```
1  #include "myfunctions.hpp"
2
3  int power(int x, int y) {
4      int result = 1;
5      for (int i = 0; i < y; ++i) result *= x;
6      return result;
7  }
```

The `root.cpp` tab is also visible, showing the following code:

```
1  #include "myfunctions.hpp"
2
3  /* return the largest z such that z^y <= x */
4  int root(int x, int y) {
5      int z = 0;
6      while (power(z + 1, y) <= x) z++;
7      return z;
8  }
```

Step 6

Create a source code file `tools/main.cpp` containing the main function which computes the 3-rd root of 74088. The result has to be printed on the screen as "The answer is" followed by the result of `root(74088,3)`.

FOLDERS

▼ power-root-example-project

▶ build

▼ include

/* myfunctions.hpp

▼ src

/* power.cpp

/* root.cpp

▼ tools

/* main.cpp

/* Makefile

◀ ▶

main.cpp

×

1

`#include <iostream>`

2

3

`#include "myfunctions.hpp"`

4

5

`int main() {`

6

`int ans = root(74088, 3);`

7

`std::cout << "The answer is " << ans << std::endl;`

8

`return 0;`

9

`}`

10

Step 7

Create a Makefile that compiles everything in an executable called theanswer:

- Specify the path of the includes using option `-I`.
- Use optimization level 3.
- Create a separate object file and a separate Makefile's target for each `.cpp` file.
- Add also a `clean` target that removes all object files.
- Optional: add other targets (e.g., `debug`) to compile with different options.

FOLDERS

- ▼ power-root-example-project
 - ▶ build
 - ▼ include
 - /* myfunctions.hpp
 - ▼ src
 - /* power.cpp
 - /* root.cpp
 - ▶ tools
- /* Makefile

```
Makefile
1  OPTIONS=-O3
2
3  all: build/theanswer
4
5  debug: OPTIONS=-O0 -g
6  debug: build/theanswer
7
8  build/theanswer: build/root.o build/power.o tools/main.cpp
9  g++ ${OPTIONS} tools/main.cpp -o build/theanswer -I include/ build/root.o build/power.o
10
11 build/root.o: src/root.cpp include/myfunctions.hpp
12 g++ ${OPTIONS} -c src/root.cpp -o build/root.o -I include/
13
14 build/power.o: src/power.cpp include/myfunctions.hpp
15 g++ ${OPTIONS} -c src/power.cpp -o build/power.o -I include/
16
17 clean:
18 rm -rf build/*.o build/theanswer
```

Step 8

- Compile and run the project.
- From the parent project folder:

```
$ make  
$ ./build/theanswer
```

or

```
$ make debug  
$ ./build/theanswer
```

Debugging — Part 1: asserts and Valgrind

The “art” of debugging

What kind of errors can we run into when programming?

- **Linking-time** errors
- **Compilation-time** errors
- **Run-time** errors – solutions:
 - debugging
 - pre/post conditions (asserts)
 - exceptions



The art of debugging

Debugging means finding **logical** errors in your program.

- Sometimes very hard to find: the program links, compiles, and runs but the **output is wrong!**
- Even worse, on some inputs the program works but on other input it does not work: hard to **reproduce** the bug.

“Reproducible bugs are candies.” — cit.

- To find bugs, we need to inspect the program at **run-time** (while it is executing).

How?

- Plan you algorithms carefully!
- Ensure pre/post conditions (asserts)
- Throw exceptions (next lectures)
- Use a debugger (Linux has **gdb**, but we will not cover this.)

Asserts — a minimalist example

```
#include <cassert>
```

```
int main() {  
    assert(3 != 3); /* throws a failed assertion */  
}
```

- `assert` is a macro (with a functional form) that takes as input a boolean condition and halts the program if the condition is not satisfied.
- **The best way to document your code.**
- Use it to check pre/post conditions for your functions.

Asserts — a minimalist example

```
#include <cassert>
```

```
int main() {  
    assert(3 != 3); /* throws a failed assertion */  
}
```

- `assert` is a macro (with a functional form) that takes as input a boolean condition and halts the program if the condition is not satisfied.
- **The best way to document your code.**
- Use it to check pre/post conditions for your functions.
- Compile with `g++ -DNDEBUG` (option `-D`, argument `NDEBUG`) to disable asserts (they are enabled by default): this way, they do not affect performance.
- Good idea to leave them enabled in debug mode, when developing a project.

Back to our example

FOLDERS

- power-root-example-project-with-asserts
 - build
 - include
 - /* myfunctions.hpp
 - src
 - /* power.cpp
 - /* root.cpp
 - tools
 - /* main.cpp
 - /* Makefile

myfunctions.hpp

```
1 #include <cassert>
2
3 int power(int, int);
4 int root(int, int);
```

include the header "cassert"

use asserts to document your code

FOLDERS

- power-root-example-project-with-asserts
 - build
 - include
 - /* myfunctions.hpp
 - src
 - /* power.cpp
 - /* root.cpp
 - tools
 - /* main.cpp
 - /* Makefile

power.cpp

```
1 #include "myfunctions.hpp"
2
3 int power(int x, int y) {
4     ... assert(y >= 0);
5     ... int result = 1;
6     ... for (int i = 0; i < y; ++i) result *= x;
7     ... return result;
8 }
9
```

main.cpp

```
1 #include <iostream>
2
3 #include "myfunctions.hpp"
4
5 int main() {
6     ... int ans = root(74088, 3);
7     ... assert(ans == 42);
8     ... std::cout << "The answer is " << ans << std::endl;
9     ... return 0;
10 }
11
```


Back to our example

disable/enable asserts

The screenshot shows a code editor with a file explorer on the left and a Makefile editor on the right. The file explorer shows a project structure with folders 'build', 'include', 'src', and 'tools'. The Makefile editor shows the following content:

```
1  OPTIONS=-O3 -DNDEBUG -Wall -Wextra
2
3  all: build/theanswer
4
5  debug: OPTIONS=-O0 -g -DDEBUG -Wall -Wextra
6  debug: build/theanswer
7
8  build/theanswer: build/root.o build/power.o tools/main.cpp
9  g++ ${OPTIONS} tools/main.cpp -o build/theanswer -I include/ build/root.o build/power.o
10
11 build/root.o: src/root.cpp include/myfunctions.hpp
12 g++ ${OPTIONS} -c src/root.cpp -o build/root.o -I include/
13
14 build/power.o: src/power.cpp include/myfunctions.hpp
15 g++ ${OPTIONS} -c src/power.cpp -o build/power.o -I include/
16
17 clean:
18 rm -rf build/*.o build/theanswer
```

Two arrows point from the text 'disable/enable asserts' to the Makefile. One arrow points to the `-DNDEBUG` flag on line 1, and the other points to the `-DDEBUG` flag on line 5.

Memory debugging

Most of C++ errors you will encounter in your code are related to (bad) **memory management**.

For example:

- Runtime errors often **abort** the program with (a rather uninformative):

segmentation fault

Often these are caused by:

- **out-of-bounds** memory accesses
 - **invalid** memory **allocation** requests
- Incorrect memory handling could result in the RAM filling up quickly (**memory leaks**).

Memory debugging

- **Warning: the compiler does detect these problems.**
- **Q. Why?**
 - Because memory monitoring can only be done at execution time: we can only detect problems while the program is running, and only if such problems occur.
 - A memory debugger simulates a machine and annotate uninitialized/unallocated memory locations.
- For memory-related problems, we will use **Valgrind** — a programming tool for memory debugging, memory leak detection, and profiling.

Valgrind

<https://valgrind.org/>

Output of
man valgrind



```
VALGRIND(1)                                Release 3.13.0                                VALGRIND(1)

NAME
    valgrind - a suite of tools for debugging and profiling programs

SYNOPSIS
    valgrind [valgrind-options] [your-program] [your-program-options]

DESCRIPTION
    Valgrind is a flexible program for debugging and profiling Linux
    executables. It consists of a core, which provides a synthetic CPU
    in software, and a series of debugging and profiling tools. The
    architecture is modular, so that new tools can be created easily
    and without disturbing the existing structure.

    Some of the options described below work with all Valgrind tools,
    and some only work with a few or one. The section MEMCHECK OPTIONS
    and those below it describe tool-specific options.

    This manual page covers only basic usage and options. For more
    comprehensive information, please see the HTML documentation on
    your system: $INSTALL/share/doc/valgrind/html/index.html, or
    online: http://www.valgrind.org/docs/manual/index.html.

TOOL SELECTION OPTIONS
    The single most important option.

    --tool=<toolname> [default: memcheck]
        Run the Valgrind tool called toolname, e.g. memcheck,
        cachegrind, callgrind, helgrind, drd, massif, lackey, none,
        exp-sgcheck, exp-bbv, exp-dhat, etc.
```

Compilation in debug mode

- In order to use Valgrind, we need to compile with `g++ -g`.
- **Q.** What does this option do?
 - It includes the names of functions, variables, etc., in the generated machine code.

This is needed because the debugger needs to tell us where the flow of execution was when the error was generated. Without these names, all the debugger could tell us would be something like:

“I’m running function 0x28fef4 and the value of variable 0x10dd12 is 42.”

- When not running in debug mode, the computer does not really care what the name of a function is (an address is enough).

Memory debugging — memcheck

We will use Valgrind's tool memcheck to detect memory bugs:

1. Use of **uninitialized memory**;
2. Reading/writing memory **after** it has been freed;
3. **Out-of-range**: reading/writing off the end of malloc'd blocks;
4. **Memory leaks**.

Uninitialized memory

```
1  #include <iostream>
2
3  int main() {
4      ... int x;
5      ... int y = x * 7;
6      ... std::cout << y << std::endl;
7      ... return 0;
8  }
9  |
```

This causes **undefined** behaviour. **Very dangerous!**

In terminal:

```
$ g++ -g uninitialized_memory_example.cpp -o build/uninitialized_memory_example
$ valgrind --track-origins=yes build/uninitialized_memory_example
```

Uninitialized memory

```
==2753==
==2753== Use of uninitialised value of size 8
==2753== at 0x4F70763: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.32)
==2753== by 0x4F70881: std::ostreambuf_iterator<char, std::char_traits<char> >
::_M_insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, st
0.32)
==2753== by 0x4F7E8F5: std::ostream& std::ostream::_M_insert<long>(long) (in /
==2753== by 0x10888A: main (uninitialized_memory_example.cpp:6)
==2753== Uninitialised value was created by a stack allocation
==2753== at 0x108865: main (uninitialized_memory_example.cpp:3)
==2753==
==2753== Conditional jump or move depends on uninitialised value(s)
==2753== at 0x4F70775: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.32)
==2753== by 0x4F70881: std::ostreambuf_iterator<char, std::char_traits<char> >
::_M_insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, st
0.32)
==2753== by 0x4F7E8F5: std::ostream& std::ostream::_M_insert<long>(long) (in /
==2753== by 0x10888A: main (uninitialized_memory_example.cpp:6)
==2753== Uninitialised value was created by a stack allocation
==2753== at 0x108865: main (uninitialized_memory_example.cpp:3)
==2753==
==2753== Conditional jump or move depends on uninitialised value(s)
==2753== at 0x4F708B6: std::ostreambuf_iterator<char, std::char_traits<char> >
::_M_insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, st
0.32)
==2753== by 0x4F7E8F5: std::ostream& std::ostream::_M_insert<long>(long) (in /
==2753== by 0x10888A: main (uninitialized_memory_example.cpp:6)
==2753== Uninitialised value was created by a stack allocation
==2753== at 0x108865: main (uninitialized_memory_example.cpp:3)
==2753==
```

```
1  #include <iostream>
2
3  int main() {
4      int x;
5      int y = x * 7;
6      std::cout << y << std::endl;
7      return 0;
8  }
9  |
```

Valgrind detects the line (6)
using the uninitialized value and
the scope (line 3) containing it.

Reading freed memory

```
1  #include <iostream>
2
3  int main() {
4      ... int* a = new int[10];
5      ... delete[] a;
6      ... std::cout << a[5] << std::endl;
7      ... return 0;
8  }
9  |
```

Often (but now always) this causes a **segmentation fault**.

In terminal:

```
$ g++ -g freed_memory_example.cpp -o build/freed_memory_example
$ valgrind --track-origins=yes build/freed_memory_example
```

Reading freed memory

Valgrind detects: where the invalid **read** is performed (6), where memory was **allocated** (4) and where it was **freed** (5).

```
1  #include <iostream>
2
3  int main() {
4      ... int* a = new int[10];
5      ... delete[] a;
6      ... std::cout << a[5] << std::endl;
7      ... return 0;
8  }
9  |
```

```
==2845== Memcheck, a memory error detector
==2845== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2845== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2845== Command: build/freed_memory_example
==2845==
==2845== Invalid read of size 4
==2845==    at 0x108926: main (freed_memory_example.cpp:6)
==2845== Address 0x5c62094 is 20 bytes inside a block of size 40 free'd
==2845==    at 0x4C3373B: operator delete[](void*) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2845==    by 0x10891D: main (freed_memory_example.cpp:5)
==2845== Block was alloc'd at
==2845==    at 0x4C3289F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2845==    by 0x108906: main (freed_memory_example.cpp:4)
==2845==
0
==2845==
```

Out-of-range errors

```
1  #include <iostream>
2
3  int main() {
4      ... int* a = new int[10];
5      ... std::cout << a[10] << std::endl;
6      ... delete[] a;
7      ... return 0;
8  }
9  |
```

Often (but now always) this causes a **segmentation fault**.

In terminal:

```
$ g++ -g invalid_read_example.cpp -o build/invalid_read_example
$ valgrind --track-origins=yes build/invalid_read_example
```

Out-of-range errors

As before, Valgrind detects: where the invalid read is performed (5) and where memory was allocated (4).

```
1  #include <iostream>
2
3  int main() {
4      ... int* a = new int[10];
5      ... std::cout << a[10] << std::endl;
6      ... delete[] a;
7      ... return 0;
8  }
9  |
```

```
==2967== Memcheck, a memory error detector
==2967== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2967== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2967== Command: build/invalid_read_example
==2967==
==2967== Invalid read of size 4
==2967==   at 0x108913: main (invalid_read_example.cpp:5)
==2967== Address 0x5c620a8 is 0 bytes after a block of size 40 alloc'd
==2967==   at 0x4C3289F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2967==   by 0x108906: main (invalid_read_example.cpp:4)
==2967==
```


Memory leaks

```
1  #include <iostream>
2
3  int main() {
4      ... int* a = new int[10];
5      ... return 0;
6  }
7  |
```

Be careful with **memory leaks**! If you repeatedly allocate memory and don't free it, you can easily fill your RAM and freeze your PC.

In terminal:

```
$ g++ -g memory_leak_example.cpp -o build/memory_leak_example
$ valgrind --track-origins=yes --leak-check=full build/memory_leak_example
```

Memory leaks

```
==2973== Memcheck, a memory error detector
==2973== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2973== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2973== Command: build/memory_leak_example
```

```
==2973==
==2973== HEAP SUMMARY:
==2973==   in use at exit: 40 bytes in 1 blocks
==2973==   total heap usage: 2 allocs, 1 frees, 73,768 bytes allocated
==2973==
==2973== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2973==   at 0x4C3289F: operator new[](unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2973==   by 0x108776: main (memory_leak_example.cpp:4)
```

```
==2973== LEAK SUMMARY:
==2973==   definitely lost: 40 bytes in 1 blocks
==2973==   indirectly lost: 0 bytes in 0 blocks
==2973==   possibly lost: 0 bytes in 0 blocks
==2973==   still reachable: 0 bytes in 0 blocks
==2973==   suppressed: 0 bytes in 0 blocks
```

```
==2973== For counts of detected and suppressed errors, rerun with: -v
==2973== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
1  #include <iostream>
2
3  int main() {
4      ... int* a = new int[10];
5      ... return 0;
6  }
7  |
```

Memory leaks

Gold rule to avoid memory leaks:

For each **new**, you have to write a corresponding **delete**.

In general: for each heap allocation, there should be a heap de-allocation (similarly, in C: for each `malloc`, you should have a `free`).

Memory leaks

- In general, don't call `delete` if you haven't called `new`.
- **Q.** Why is not (a manual) `delete` necessary instead in the following piece of code?

```
#include <vector>
```

```
int main() {
```

```
    auto v = std::vector<int>(100);
```

```
}
```

- Because `v` is on the stack (not on the heap). The object's destructor is automatically called when `v` is removed from the stack (i.e., when exiting `v`' scope).

Memory debugging

To sum up, one command for everything (note the compilation option `-g`):

```
$ g++ -g test.cpp -o test  
$ valgrind --leak-check=full --track-origins=yes ./test
```

What you want is that the output of Valgrind ends with:

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Q. But why should you really care about memory bugs?

Exam project

You



me finding a
memory leak
in your project

teaching
assistants

Debugging — Part 2: compiler warnings and sanitizers

Compiler warnings

- The compiler can be instructed to report **warnings**.
- A warning is not an error but a suggestion that something is probably missing in your code or might go wrong...
- Comprehensive list of compiler flags to enable warnings:
<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>
- We will use two of them: `-Wall` `-Wextra`

`-Wall`

`-Wextra`

`-Wclobbered`
`-Wcast-function-type`
`-Wdeprecated-copy`
`-Wempty-body`
`-Wenum-conversion`
`-Wignored-qualifiers`
`-Wimplicit-fallthrough=3`
`-Wmissing-field-initializers`
`-Wmissing-parameter-type`
`-Wold-style-declaration`
`-Woverride-init`
`-Wsign-compare`
`-Wstring-compare`
`-Wredundant-move`
`-Wtype-limits`
`-Wuninitialized`
`-Wshift-negative-value`
`-Wunused-parameter`
`-Wunused-but-set-parameter`

`-Waddress`
`-Warray-bounds=1`
`-Warray-compare`
`-Warray-parameter=2`
`-Wbool-compare`
`-Wbool-operation`
`-Wc++11-compat-Wc++14-compat`
`-Wcatch-value`
`-Wchar-subscripts`
`-Wcomment`
`-Wdangling-pointer=2`
`-Wduplicate-decl-specifier`
`-Wenum-compare`
`-Wenum-int-mismatch`
`-Wformat`
`-Wformat-overflow`
`-Wformat-truncation`
`-Wint-in-bool-context`
`-Wimplicit`
`-Wimplicit-int`
`-Wimplicit-function-declaration`
`-Winit-self`
`-Wlogical-not-parentheses`
`-Wmain`
`-Wmaybe-uninitialized`
`-Wmemset-elt-size`
`-Wmemset-transposed-args`
`-Wmisleading-indentation`
`-Wmismatched-dealloc`
`-Wmismatched-new-delete`
`-Wmissing-attributes`
`-Wmissing-braces`
`-Wmultistatement-macros`
`-Wnarrowing`
`-Wnonnull`
`-Wnonnull-compare`
`-Wopenmp-simd`
`-Wparentheses`
`-Wpessimizing-move`
`-Wpointer-sign`
`-Wrange-loop-construct`
`-Wreorder`
`-Wrestrict`
`-Wreturn-type`
`-Wself-move`
`-Wsequence-point`
`-Wsign-compare`
`-Wsizeof-array-div`
`-Wsizeof-pointer-div`
`-Wsizeof-pointer-memaccess`
`-Wstrict-aliasing`
`-Wstrict-overflow=1`
`-Wswitch`
`-Wtautological-compare`
`-Wtrigraphs`
`-Wuninitialized`
`-Wunknown-pragmas`
`-Wunused-function`
`-Wunused-label`
`-Wunused-value`
`-Wunused-variable`
`-Wuse-after-free=3`
`-Wvla-parameter`
`-Wvolatile-register-var`
`-Wzero-length-bounds`

Compiler warnings — Example

```
1  #include <iostream>
2
3  int main() {
4      int x;
5      int y = x * 7;
6      std::cout << y << std::endl;
7  }
```

```
→ Desktop g++ test.cpp -o test -Wall -Wextra
test.cpp:5:11: warning: variable 'x' is uninitialized when used here [-Wuninitialized]
    int y = x * 7;
            ^
test.cpp:4:8: note: initialize the variable 'x' to silence this warning
    int x;
        ^
        = 0
1 warning generated.
→ Desktop █
```

Always add these two flags in your Makefile

FOLDERS

- ▼ power-root-example-project-with-asserts
 - ▶ build
 - ▶ include
 - ▶ src
 - ▶ tools
- /* Makefile

Makefile

```
1  OPTIONS=-O3 -DNDEBUG -Wall -Wextra
2
3  all: build/theanswer
4
5  debug: OPTIONS=-O0 -g -DDEBUG -Wall -Wextra
6  debug: build/theanswer
7
8  build/theanswer: build/root.o build/power.o tools/main.cpp
9  |—— g++ ${OPTIONS} tools/main.cpp -o build/theanswer -I include/ build/root.o build/power.o
10
11 build/root.o: src/root.cpp
12 |—— g++ -c src/root.cpp -o build/root.o -I include/
13
14 build/power.o: src/power.cpp
15 |—— g++ -c src/power.cpp -o build/power.o -I include/
16
17 clean:
18 |—— rm -rf build/*.o build/theanswer
```

Introducing compiler sanitizers

- Valgrind is a great tool to hunt memory bugs down. **Use it!**
- But Valgrind is a **simulator**: there could be some subtle problems in the code that look fine for Valgrind.
- Valgrind **does not scale**. If you are working on a large project and with some large datasets (large memory allocations), it will slow down program execution by (typically) one order to magnitude (10 times slower).
- Thus, Valgrind is fine for developing/debugging small-medium size projects.

Introducing compiler sanitizers

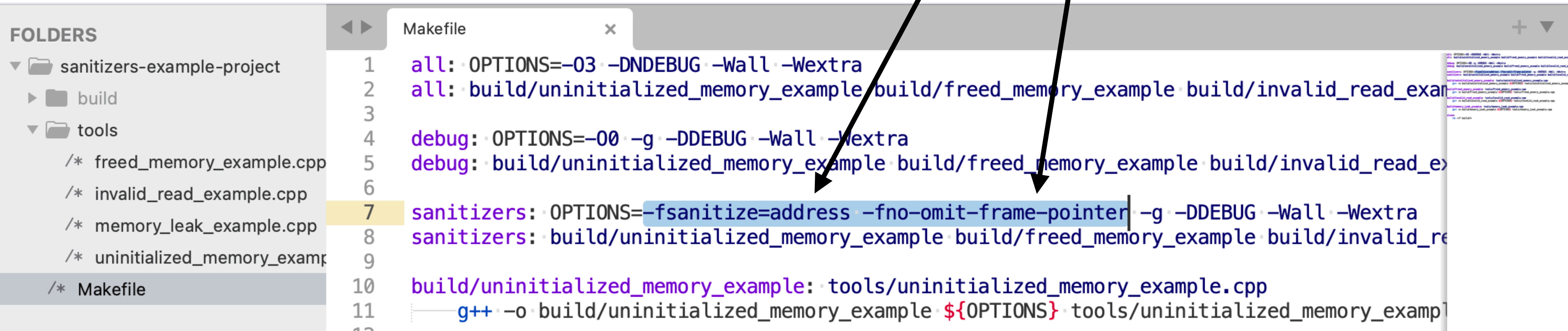
- Valgrind is a great tool to hunt memory bugs down. **Use it!**
- But Valgrind is a **simulator**: there could be some subtle problems in the code that look fine for Valgrind.
- Valgrind **does not scale**. If you are working on a large project and with some large datasets (large memory allocations), it will slow down program execution by (typically) one order to magnitude (10 times slower).
- Thus, Valgrind is fine for developing/debugging small-medium size projects.
- There exists a more precise alternative: **compiler sanitizers**.
- Two more compiler flags: `-fsanitize=address` and `-fno-omit-frame-pointer`.

<https://clang.llvm.org/docs/AddressSanitizer.html>

Compiler sanitizers

- Create a new target “sanitizers” in your Makefile with these two flags.
- You can enable/disable sanitizers at compile time:

```
$ make clean; make  
$ make clean; make sanitizers
```



Example 1: invalid read

FOLDERS

▼

 sanitizers-example-project

▶

 build

▼

 tools

/* freed_memory_example.cpp

/* invalid_read_example.cpp

/* memory_leak_example.cpp

/* uninitialized_memory_example.

/* Makefile

invalid_read_example.cpp

×

1

#include <iostream>

2

3

int main() {

4

int *a = new int[10];

5

std::cout << a[10] << std::endl;

6

delete[] a;

7

return 0;

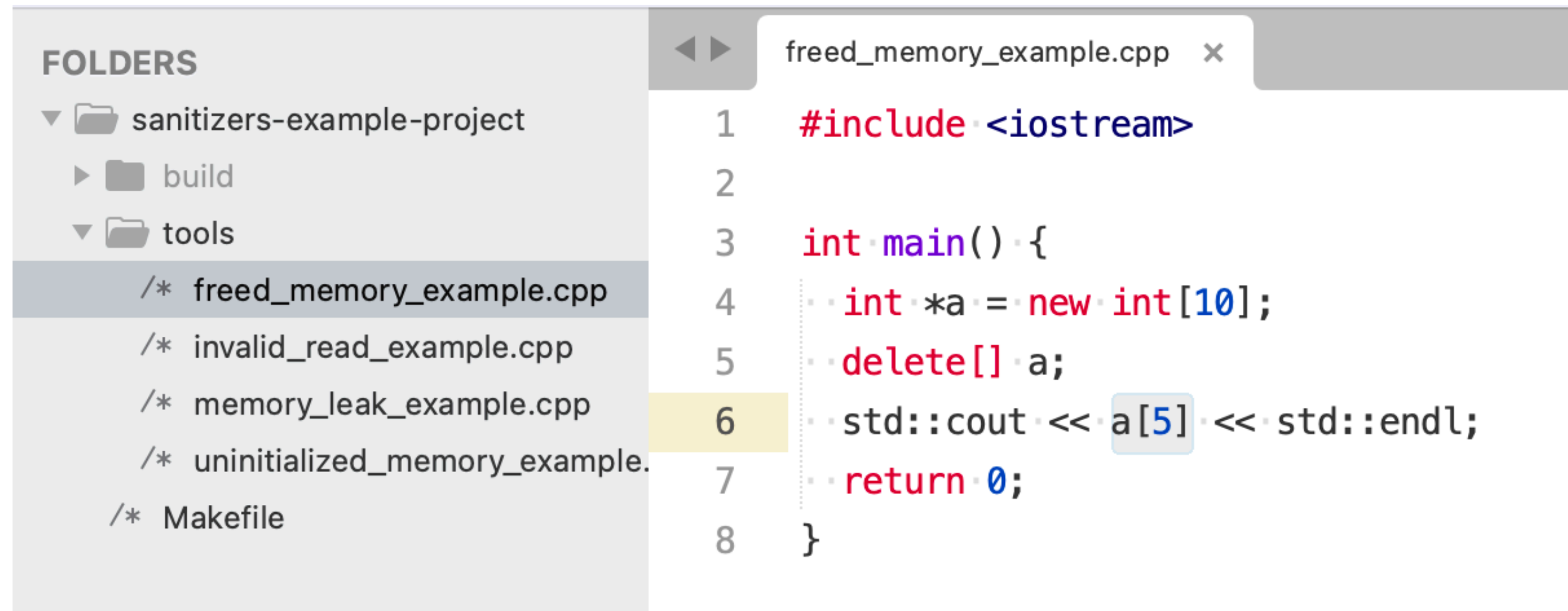
8

}

```
==654763==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x604000000038 at pc 0x562d9e833322 bp 0x7ffe3a665210 sp 0x7ffe3a665200
READ of size 4 at 0x604000000038 thread T0
#0 0x562d9e833321 in main tools/invalid_read_example.cpp:5
#1 0x7f30277a6fcf in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
#2 0x7f30277a707c in __libc_start_main_impl ../csu/libc-start.c:409
#3 0x562d9e833204 in _start (/home/giulio/pel/slides/1-introduction/code/sanitizers-example-project/build/invalid_read_example+0x1204)

0x604000000038 is located 0 bytes to the right of 40-byte region [0x604000000010,0x604000000038)
allocated by thread T0 here:
#0 0x7f3027d72337 in operator new[](unsigned long) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:102
#1 0x562d9e8332de in main tools/invalid_read_example.cpp:4
#2 0x7f30277a6fcf in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
```


Example 2: accessing freed memory



```
FOLDERS
▼ sanitizers-example-project
  ► build
  ▼ tools
    /* freed_memory_example.cpp
    /* invalid_read_example.cpp
    /* memory_leak_example.cpp
    /* uninitialized_memory_example.cpp
    /* Makefile

freed_memory_example.cpp x
1  #include <iostream>
2
3  int main() {
4      int *a = new int[10];
5      delete[] a;
6      std::cout << a[5] << std::endl;
7      return 0;
8  }
```

```
==654728==ERROR: AddressSanitizer: heap-use-after-free on address 0x604000000024 at pc 0x55f6d70a6335 bp 0x7fff3f1671e0 sp 0x7fff3f1671d0
READ of size 4 at 0x604000000024 thread T0
```

```
#0 0x55f6d70a6334 in main tools/freed_memory_example.cpp:6
#1 0x7faeb0877fcf in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
#2 0x7faeb087807c in __libc_start_main_impl ../csu/libc-start.c:409
#3 0x55f6d70a6204 in _start (/home/giulio/pel/slides/1-introduction/code/sanitizers-example-project/build/freed_memory_example+0x1204)
```

```
0x604000000024 is located 20 bytes inside of 40-byte region [0x604000000010,0x604000000038)
```

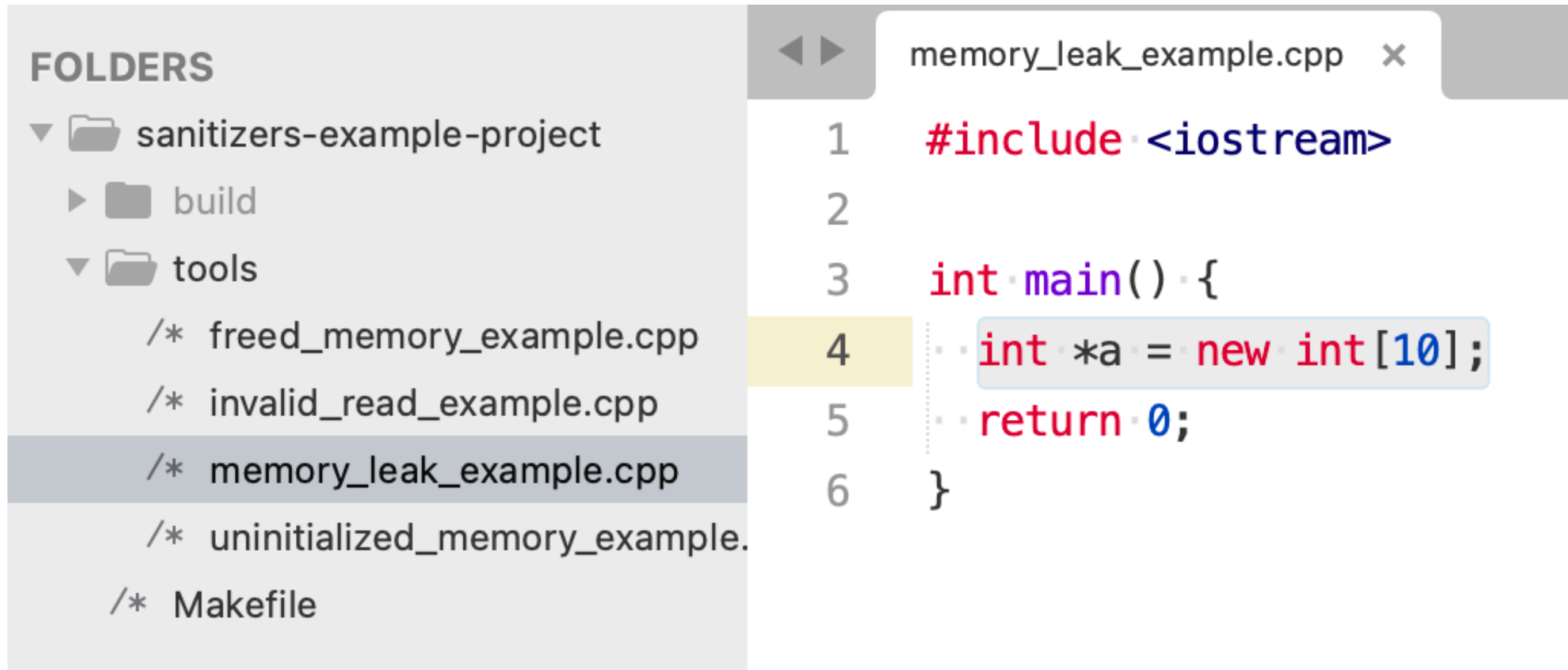
```
freed by thread T0 here:
```

```
#0 0x7faeb0e43e37 in operator delete[](void*) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:163
#1 0x55f6d70a62f5 in main tools/freed_memory_example.cpp:5
#2 0x7faeb0877fcf in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
```

```
previously allocated by thread T0 here:
```

```
#0 0x7faeb0e43337 in operator new[](unsigned long) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:102
#1 0x55f6d70a62de in main tools/freed_memory_example.cpp:4
#2 0x7faeb0877fcf in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
```

Example 3: memory leaks



FOLDERS

- ▼ sanitizers-example-project
 - ▶ build
 - ▼ tools
 - /* freed_memory_example.cpp
 - /* invalid_read_example.cpp
 - /* memory_leak_example.cpp
 - /* uninitialized_memory_example.
 - /* Makefile

```
memory_leak_example.cpp x
1  #include <iostream>
2
3  int main() {
4      int *a = new int[10];
5      return 0;
6  }
```

==657730==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 40 byte(s) in 1 object(s) allocated from:

```
#0 0x7f9af29e6337 in operator new[](unsigned long) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:102
#1 0x56106d87225e in main tools/memory_leak_example.cpp:4
#2 0x7f9af241afcf in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
```

Sanitizers vs. Valgrind

- **Make your code safer by compiling with sanitizers when developing.**
- Valgrind is an external tool; not shipped with your compiler.
- Sanitizers work at the **compilation level**: you need the source code.
Valgrind is a simulator and you only need the executable.
- This also means that to disable sanitizers, you have to compile it again.
With Valgrind: compile once and use it only when needed.
- Sanitizers are more precise and can detect more errors.
- Sanitizers do not slow down your program by one order of magnitude (typically, you can expect a slowdown of 2X); Valgrind does.
- Unfortunately, sanitizers are not well supported on Mac (hopefully, support will come in the close future).

Use both for your project!