Compilation

Victor Eijkhout, Susan Lindsey

Fall 2025

last formatted: October 14, 2025

Intro to file types

1. File types

Text files	
Source	Program text that you write
Header	also written by you, but not really program text.
Binary files	
Object file	The compiled result of a single source file
Library	Multiple object files bundled together
Executable	Binary file that can be invoked as a command
Data files	Written and read by a program

2. Text files

- Source files and headers
- You write them: *make sure you master an editor*
- The computer has no idea what these mean.
- They get compiled into programs.

(Also 'just text' files: READMEs and such)

3. Binary files

- Programs. (Also: object and library files.)
- Produced by a compiler.
- Unreadable by you; executable by the computer.

Also binary data files; usually specific to a program.
(Why don't programs write out their data in readable form?)

Compilation

4. Compilers

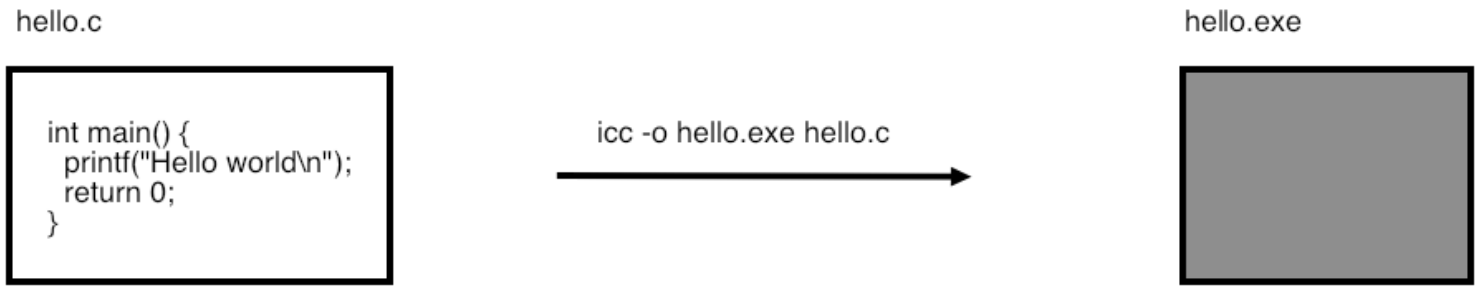
Compilers: a major CS success story.

- The first Fortran compiler (Backus, IBM, 1954): multiple man-years.
- These days: semester project for graduate students. Many tools available (`lex`, `yacc`, `clang-tidy`) Standard textbooks ('Dragon book')
- Compilers are very clever!
  You can be a little more clever in assembly – maybe but compiled languages are 10× more productive.

5. Compilation vs interpreted

- Interpreted languages: lines of code are compiled 'just-in-time'. Very flexible, sometimes very slow.
- Compiled languages: code is compiled to machine language: less flexible, very fast execution.
- Virtual machine: languages get compiled to an intermediate language
  (Pascal, Python, Java)
  pro: portable; con: does not play nice with other languages.
- Scientific computing languages:
  - Fortran: pretty elegant, great at array manipulation
    Note: Fortran20003 is modern; F77 and F90 are not so great.
  - C: low level, allows great control, tricky to use
  - C++: allows much control, more protection, more tools
    (kinda sucks at arrays)

6. Simple compilation



- From source straight to program.
- Use this only for short programs.

```
%% icpx hello.c
%% ./a.out
hello world
```

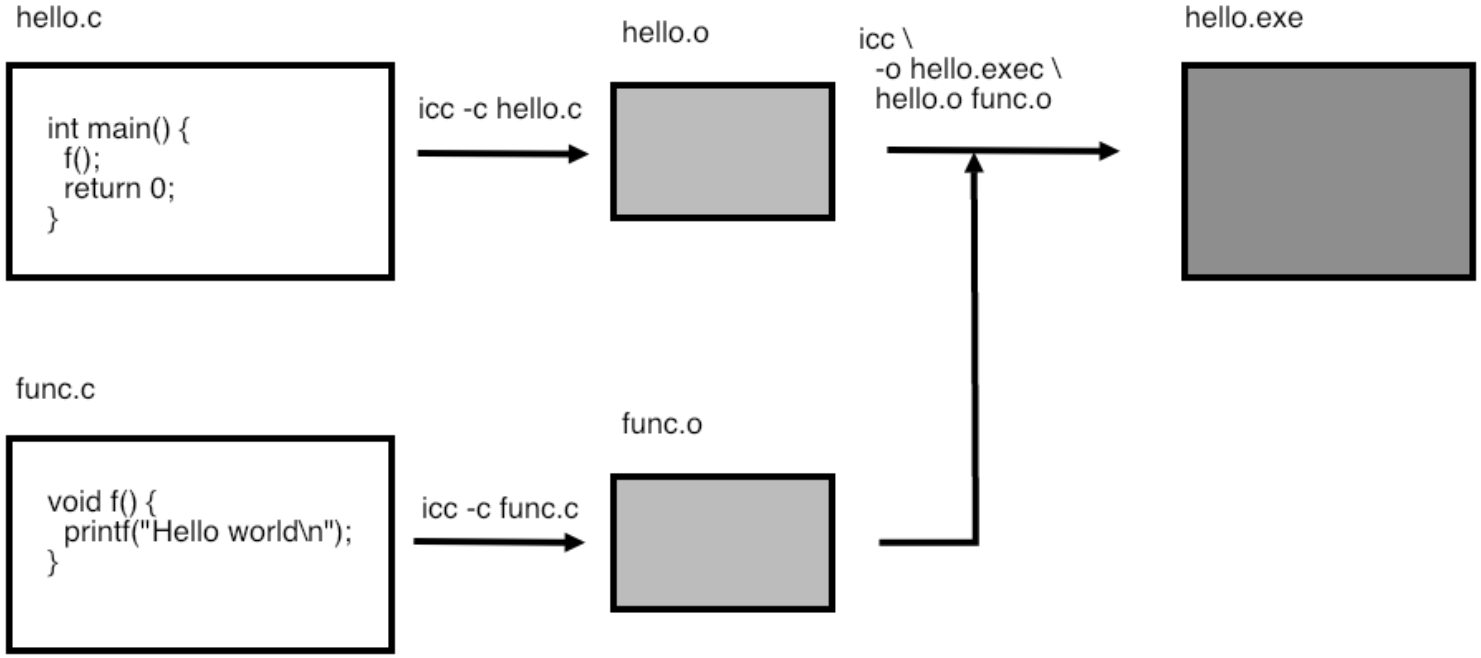
```
%% icpx -o helloprog hello.c
%% ./helloprog
hello world
```

7. Exercise 1

Create a file with these contents, and make sure you can compile it:

```
1 #include <iostream>
2 using std::cout;
3
4 int main() {
5   cout << "hello world\n";
6   return 0;
7 }
```

8. Separate compilation



- Large programs best broken into small files,
- ... and compiled separately (can you guess why?)
- Then 'linked' into a program; linker is usually the same as the compiler.

9. Exercise 2

Make the following files:

Main program: `fooprog.cpp`

```
1 #include <iostream>
2 using std::cout;
3 #include <string>
4 using std::string;
5
6 extern void bar(string);
7
8 int main() {
9   bar("hello world\n");
10  return 0;
11 }
```

Subprogram: `foosub.cpp`

```
1 #include <iostream>
2 using std::cout;
3 #include <string>
4 using std::string;
5
6 void bar( string s ) {
7   cout << s << '\n';
8 }
```

10. Exercise 2 continued

- Compile in one:

  ```
  icpx -o program fooprog.c foosub.c
  ```

- Compile in steps:

  ```
  icpx -c fooprog.c
  icpx -c foosub.c
  icpx -o program fooprog.o foosub.o
  ```

  What files are being produced each time?

Can you write a shell script to automate this?

11. Header files

- `extern` is not the best way of dealing with 'external references'
- Instead, make a header file `foo.h` that only contains

```
1 void bar(string);
```

- Include it in both source files:

```
1 #include "foo.h"
```

- Do the separate compilation calls again.

Now is a good time to learn about makefiles ...

12. Compiler options 101

- You have just seen two compiler options.
- Commandlines look like

  ```
  command [ options ] [ argument ]
  ```

  where square brackets mean: 'optional'
- Some options have an argument

  ```
  icpx -o myprogram mysource.c
  ```
- Some options do not.

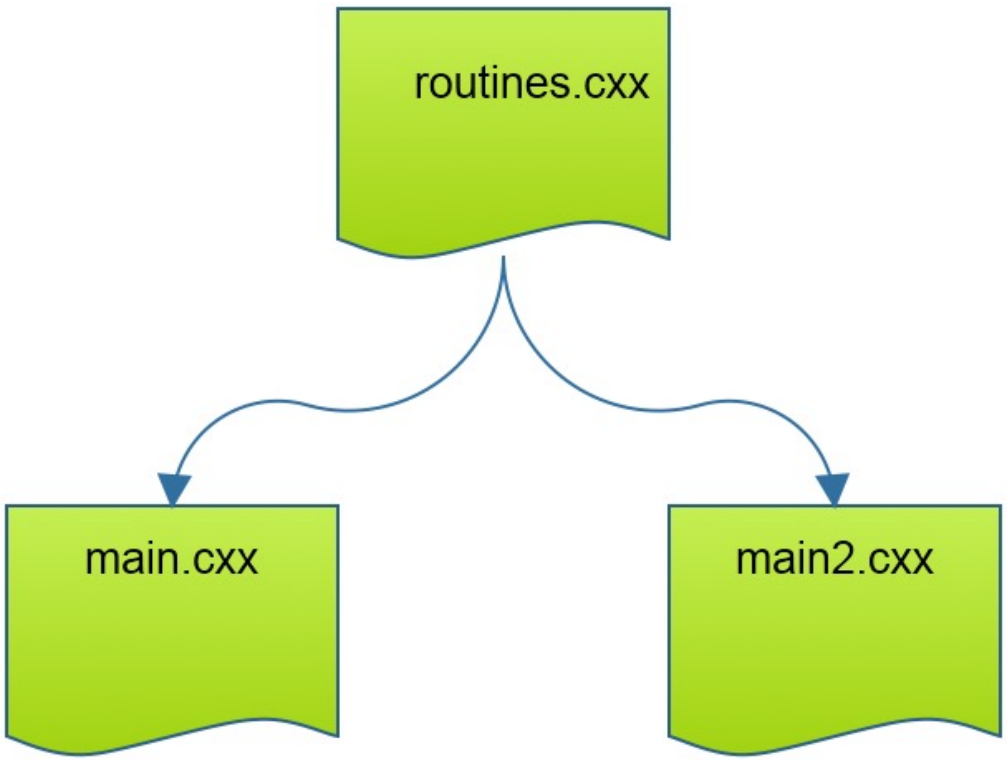
  ```
  icpx -g -o myprogram mysource.c
  ```
- Question: does -c have an argument? How can you find out?

  ```
  icpx -g -c mysource.c
  ```

Separate compilation

13. Include files

- Code reuse is good.
- How would you use functions/classes in more than one main?



We will discuss systematic solutions.

14. Reminder: definition vs declaration

Definition:

```
1     bool iseven( int n ) { return n%2==0 }
2
```

Declaration:

```
1     bool iseven( int n );
2     // or even:
3     bool iseven( int );
4
```

15. Declarations, case 1

Some people like defining functions after the main.
Problem: the main needs to know about them.

'forward declaration'

```
1  int f(int);
2  int main() {
3    f(5);
4  };
5  int f(int i) {
6    return i;
7  }
```

versus:

```
1 int f(int i) {
2   return i;
3 }
4 int main() {
5   f(5);
6 };
```

This is a stylistic choice.

16. Declarations, case 2

You also need forward declaration for mutually recursive functions:

```
1 int f(int);
2 int g(int i) { return f(i); }
3 int f(int i) { return g(i); }
```

17. Separate compilation

Split your program in multiple files.

- Easier to edit
- Less chance of `git` conflicts
- Only recompile the file you edit
  $\Rightarrow$ reduction of compile/build time.

18. Declarations for separate compilation

Define a function in one file;
an other file uses it, so needs the declaration:

```
1 // file: def.cpp
2 int tester(float x) {
3   .....
4 }
```

```
1 // file : main.cpp
2 int tester(float);
3
4 int main() {
5   int t = tester(...);
6   return 0;
7 }
```

This Is Not A Good Design!

19. Declarations and header files

Using a header file with function declarations.

Header file contains only declaration:

```
1 // file: def.h
2 int tester(float);
```

The header file gets included twice.

Definitions file:

```
1 // compile/testerdef.cpp
2 #include "def.h"
3 int tester(float x) {
4   .....
5 }
```

Main program:

```
1 // compile/testermain.cpp
2 #include "def.h"
3 int main() {
4   int t = tester(...);
5   return 0;
6 }
```

What happens in both cases if you leave out the `#include "def.h"`?

20. Class declarations

Header file:

```
1 // proto/header.hpp
2 class something {
3 private:
4   int i;
5 public:
6   double dosomething( int i, char c );
7 };
```

Implementation file:

```
1 // proto/func.cpp
2 double something::dosomething( int i, char c ) {
3   // do something with i,c
4 };
```

21. File naming convention

- Source files: `.cpp` `.cxx`
  I use `.cpp` for no real reason
- Header files: `.h` `.hpp` `.hxx`
  I use `.hpp` by analogy with `.cpp`
  `.h` reminds me too much of C.

22. Compiling and linking

Your regular compile line

```
1 icpc -o yourprogram yourfile.cpp
```

actually does two things: compilation, and linking. You can do those separately:

1. First you compile

   ```
   1 icpc -c yourfile.cpp
   ```

   which gives you a file `yourfile.o`, a so-called object file; and
2. Then you use the compiler as linker to give you the executable file:

   ```
   1 icpc -o yourprogram yourfile.o
   ```

23. Dealing with multiple files

Compile each file separately, then link:

```
1 icpc -c mainfile.cc
2 icpc -c functionfile.cc
3 icpc -o yourprogram mainfile.o functionfile.o
```

24. Declarations and header files

Using a header file with function declarations.

Header file contains only declaration:

```
1 // file: def.h
2 int tester(float);
```

The header file gets included twice.

Definitions file:

```
1 // compile/testerdef.cpp
2 #include "def.h"
3 int tester(float x) {
4   .....
5 }
```

Main program:

```
1 // compile/testermain.cpp
2 #include "def.h"
3 int main() {
4   int t = tester(...);
5   return 0;
6 }
```

What happens in both cases if you leave out the `#include "def.h"`?

25. Class declarations

Header file:

```
1 // proto/header.hpp
2 class something {
3 private:
4   int i;
5 public:
6   double dosomething( int i, char c );
7 };
```

Implementation file:

```
1 // proto/func.cpp
2 double something::dosomething( int i, char c ) {
3   // do something with i,c
4 };
```

26. Header file with include guard

Header file tests if it has already been included:

```
1 // this is foo.h
2 #ifndef FOO_H
3 #define FOO_H
4
5 // the things that you want to include
6
7 #endif
```

This prevents double or recursive inclusion.

Quiz 1

For each of the following answer: is this a valid function definition or function declaration.

- `int foo();`
- `int foo() {};`
- `int foo(int) {};`
- `int foo(int bar) {};`
- `int foo(int) { return 0; };`
- `int foo(int bar) { return 0; };`

27. Make

Good idea to learn the `Make` utility for project management.

(Also `Cmake`.)

28. Skeleton example

Directory `skeletons/funct_skeleton` contains

```
funct.cpp functheader.hpp functmain.cpp
```

CMake setup:

```
add_executable(
    funct functmain.cpp funct.cpp functheader.hpp )
```

29. CMake compilation

Do `cmake` and then `make`:

```
[ 33%] Building CXX object CMakeFiles/funct.dir/functmain.cpp.o
[ 66%] Building CXX object CMakeFiles/funct.dir/funct.cpp.o
[100%] Linking CXX executable funct
[100%] Built target funct
```

30. Justification for separate compilation

- Edit only `funct.cpp`;
- Do *not* `cmake`;
- do `make`

```
( cd build && make )
Consolidate compiler generated dependencies of target funct
[ 33%] Building CXX object CMakeFiles/funct.dir/funct.cpp.o
[ 66%] Linking CXX executable funct
[100%] Built target funct
```

Only that file got recompiled.

More about compilation

31. Object files

- Object files are unreable. (Try it. How do you normally view files? Which tool sort of works?)
- But you can get some information about them.

```
#include <stdlib.h>
#include <stdio.h>
void bar(char *s) {
  printf("%s",s);
}
```

```
[c:264] nm foosub.o
0000000000000000 T _bar
U _printf
```

Where `T`: stuff defined in this file
`U`: stuff used in this file

32. Compiler options 102

- Optimization level: -O0, -O1, -O2, -O3
  ('I compiled my program with oh-two')
  Higher levels usually give faster code. Level 3 can be unsafe. (Why?)
- -g is needed to run your code in a debugger. Always include this.
- The ultimate source is the 'man page' for your compiler.

33. Compiler optimizations

Common subexpression elimination:

```
1 x1 = pow(5.2,3.4) * 1;
2 x2 = pow(5.2,3.4) * 2;
```

becomes

```
1 t = pow(5.2,3.4);
2 x1 = t * 1;
3 x2 = t * 2;
```

Loop invariants lifting

```
1 for (int i=0; i<1000; i++)
2   s += 4*atan(1.0) / i;
```

becomes

```
1 t = 4*atan(1.0);
2 for (int i=0; i<1000; i++)
3   s += t / i;
```

34. Example of optimization

Givens program Run with optimization level 0,1,2,3 we get:

```
Done after 8.649492e-02
Done after 2.650118e-02
Done after 5.869865e-04
Done after 6.787777e-04
```