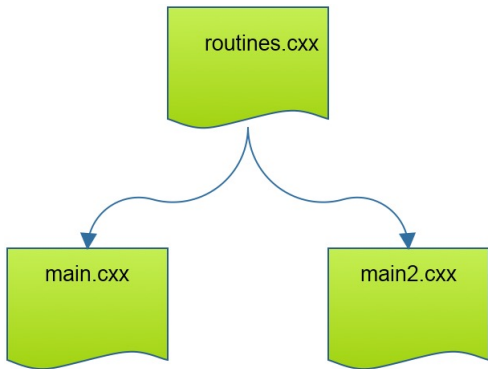# Prototypes

Victor Eijkhout, Susan Lindsey

Fall 2023
last formatted: October 19, 2023

1. **Include files**

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

We will discuss systematic solutions.

2. Reminder: definition vs declaration

Definition:

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

Declaration:

```
bool iseven( int n );
// or even:
bool iseven( int );
```

3. Declaractions, case 1

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

'forward declaration'

```
int f(int);
int main() {
  f(5);
};
int f(int i) {
  return i;
}
```

versus:

```
int f(int i) {
  return i;
}
int main() {
  f(5);
};
```

This is a stylistic choice.

4. Declarations, case 2

You also need forward declaration for mutually recursive functions:

```
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

5. 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.

6. Declarations for separate compilation

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

```cpp
// file: def.cpp
int tester(float x) {
  .....
}
```

```cpp
// file : main.cpp
int tester(float);

int main() {
  int t = tester(...);
  return 0;
}
```

This Is Not A Good Design!

7. Declarations and header files

Using a header file with function declarations.

Header file contains only declaration:

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

The header file gets included both in the definitions file and the main program:

```
// file: def.cpp
#include "def.h"
int tester(float x) {
  .....
}
```

```
// file : main.cpp
#include "def.h"

int main() {
  int t = tester(...);
  return 0;
}
```

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

8. Class declarations

Header file:

```
// proto/functheader.hpp
class something {
private:
  int i;
public:
  double dosomething( int i, char c );
};
```

Implementation file:

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

9. File naming convention

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

10. Compiling and linking

Your regular compile line

```
icpc -o yourprogram yourfile.cc
```

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

1. First you compile

   ```
   icpc -c yourfile.cc
   ```

   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:

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

11. **Dealing with multiple files**

Compile each file separately, then link:

```
icpc -c mainfile.cc
icpc -c functionfile.cc
icpc -o yourprogram mainfile.o functionfile.o
```

12. Header file with include guard

Header file tests if it has already been included:

```
// this is foo.h
#ifndef FOO_H
#define FOO_H

// the things that you want to include

#endif
```

Prevent double or recursive inclusion.

13. Make

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

(Also `Cmake`.)

14. **Skeleton example**

Directory skeletons/funct_skeleton contains

funct.cpp functheader.hpp functmain.cpp

CMake setup:

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

15. **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
```

16. 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.