# Prototypes

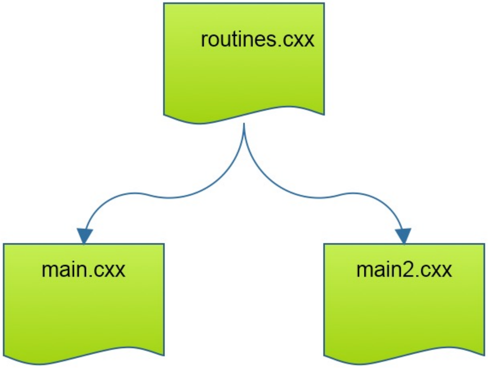
Victor Eijkhout, Susan Lindsey

Fall 2023

`last formatted: October 19, 2023`

## 1. Include files

Reuse code by include it in multiple mains.



We will develop a better scenario.

## 2. Forward declarations, 1

A first use of declarations is forward declarations.

Some people like defining functions after the main:

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

versus before:

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

## 3. Forward declarations, 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); }
```

## 4. Declarations for separate compilation

Declare a function in one file
make it known in another

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

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

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

This Is Not A Good Design!

## 5. 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?

## 6. 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
   ```

## 7. 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
```

## 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. 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.

## 10. Make

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

(Also `Cmake`.)

## 11. Skeleton example

Directory `skeletons/funct_skeleton` contains

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

CMake setup:

## 12. CMake compilation

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

## 13. Justification for separate compilation

You edit only `funct.cpp`; then

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

Only that file got recompiled.