CSE142
Computer Programming I

Structuring Program Files

… or Which came first? The prototype or the definition?

G3-1
4/16/01

Structuring Programs

Programs often use many functions defined locally and borrowed from libraries.

Organizing functions (and other parts) within and among .c and .h files is important:

- lets compiler understand how code fits together
- groups logically connected sets of behavior
- allows programmers to separate implementation of behavior from its specification

G3-2
4/16/01

Big Brother is Watching

Most C compilers will tell you if you call a function (or use a variable!) improperly:

- too many/few arguments
- trying to use value of a void function
- passing an argument to a parameter with an incompatible type

*How does it **know** when to warn you?*
What does it need to give these warnings?

G3-3
4/16/01

Order in the Program

General principle: identifiers (names) must be declared before they are used.

- For variables, this means:

  place them first within a function

- For symbolic constants (#defined stuff):

  place them at the top of the file

- For functions:

  declare them before they are called

G3-4
4/16/01

Order for Functions in a .c File

Function names are identifiers, so… they too must be declared before they are used:

```
#include <stdio.h>
void fun2  (void) { ... }
void fun1  (void) { ...; fun2( ); ... }
int   main (void) { ...; fun1( ); ... return 0; }
```

fun1 calls fun2: so, fun2 is defined before fun1.

main calls fun1: so, fun1 is defined before main.

G3-5
4/16/01

A Tangled Web

Insisting that each function *entirely* precede any calls to it can be annoying:

- **frustrating**: write the niggly little functions at the top and the important ones at the bottom
- **inconvenient**: printf is a function, but we don't want its code in our program!
- **impossible**: function A calls function B and function B also calls function A

Is there any solution?
Can anyone help us?

G3-6
4/16/01

Look, Up in the Air: Function Prototypes

Function prototypes allow us to declare the function's name without giving its code.

Now we can use it before fully defining it!

In particular, the prototype gives:

- the name of the function
- the return type of the function
- the types of all the function's parameters

G3-7
4/16/01

Prototype Syntax

return_type name(type parm1, type parm2, …);

(…like a function with "**;**" instead of "**{…}**"!)

Examples:

- void Useless(void);
- void PrintInteger(int value);
- double CalculateTax(double amount,
  double rate);

Is this enough to call the function?
*Bonus: is this enough to **understand** the function?*

Using Prototypes

Write prototypes for all your functions near the top of the program.

- You can call the function *anywhere* thereafter!

Fully define the function later, wherever it fits logically.

*This is **not** required by C.*
But… it's highly recommended to organize and elucidate your program.

G3-9
4/16/01

Structuring Programs

Programs often use many functions defined locally and borrowed from libraries.

Organizing functions (and other parts) within and among .c and .h files is important:

- lets compiler understand how code fits together
- groups logically connected sets of behavior
- allows programmers to separate implementation of behavior from its specification

G3-10
4/16/01

Libraries

Question:

What about library functions, like printf?
Does the compiler need *their* prototype and code?

Answer: that is the purpose of the #include directive:

- #include gets printf's prototype for the compiler
- the linker knows where its body (code) is

G3-11
4/16/01

#include <stdio.h>

The "#include <foo.h>" means:

"get the file foo.h and insert what's in it right here (as if it had been typed here)"

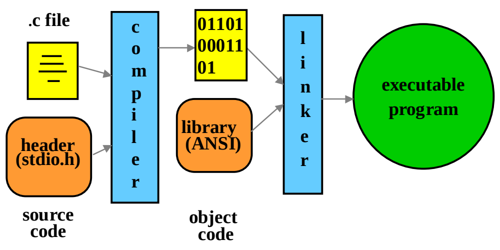
stdio.h contains prototypes for scanf, printf, and the other functions in the standard I/O library

Their implementations (bodies) are **NOT** there!

The code for these functions is in a library that is combined with your code by the linker.

*So, prototypes enable grouping behaviors and separating code & spec. **You** can do this, too! (Not with .h files, for now.)*

Compilers and Linkers and Executables



G3-13
4/16/01

Putting It All Together

#include directives

…

#define constants

…

Function prototypes

…

Full function definitions

…

G3-14
4/16/01

Logical Order vs. Control Flow

With prototypes, your functions can be placed in any physical order.

Order within the source file has *no influence* on control flow.

Programs always start executing at the function main.

(So, there should always be a main.)

No function is executed until it is called by some other function (except main).

G3-15
4/16/01

Summary

- Organizing the parts of a .c file is important
- General principle: Identifiers must be declared before they are used.
- For functions, a prototype can be declared:
  - Prototype: near the beginning of the program
  - Function detail: later on
- Prototypes allow us to group behaviors logically and separate implementation from specification.
- Source order and control flow are different concepts

G3-16
4/16/01

QOTD (early): A Need to Know Basis

Functions tie together a lot of information: return type, name, parameter types, parameter names, parameter order, number of parameters, and body.

Which of these aspects of functions should each of the following *need to know*?

- the body of the function
- someone trying to use the function
- the compiling and linking processes (together)

In other words: which aspects would each of these "parties" need to find out about if they changed?

G3-17
4/16/01