COP3514 Program Design

Spring 2022

Instructor: Mauricio Pamplona Segundo

Agenda

- Modularity
- Header Files
- Dividing a Program into Files

Chapter 15

Writing large programs

Modules

- It's often useful to view a program as a number of independent ***modules.***

- A module is a collection of services, some of which are made available to other parts of the program (the ***clients***).
  - For example, a module containing I/O functions.

Modules

- Each module has an ***interface*** that describes the available services.
  - For example, function prototypes of scanf, printf, fscanf, …

- The details of the module—including the source code for the services themselves—are stored in the module's ***implementation.***

Modules

- In the context of C, “services” are functions.
- The interface of a module is a header file containing prototypes for the functions that will be made available to clients (source files).
- The implementation of a module is a source file that contains definitions of the module’s functions.

Modules

- The C library is itself a collection of modules.

- Each header in the library serves as the interface to a module.
  - `<stdio.h>` is the interface to a module containing I/O functions, `stdio.c` is the implementation.
  - `<string.h>` is the interface to a module containing string-handling functions, `string.c` is the implementation.

Modules

- Advantages of dividing a program into modules:
  - Abstraction
  - Reusability
  - Maintainability

Modules

- ***Abstraction.*** A properly designed module can be treated as an ***abstraction;*** we know what it does, but we don't worry about how it works.

- Thanks to abstraction, it's not necessary to understand how the entire program works in order to make changes to one part of it.

- Abstraction also makes it easier for several members of a team to work on the same program.

Modules

- ***Reusability.*** Any module that provides services is potentially reusable in other programs.

- Since it’s often hard to anticipate the future uses of a module, it’s a good idea to design modules for reusability.

Modules

- ***Maintainability.*** A small bug will usually affect only a single module implementation, making the bug easier to locate and fix.

- Rebuilding the program requires only a recompilation of the module implementation (followed by linking the entire program).

- An entire module implementation can be replaced if necessary.

Modules

- Maintainability is the most critical advantage.
- Most real-world programs are in service over a period of years
- During this period, bugs are discovered, enhancements are made, and modifications are made to meet changing requirements.
- Designing a program in a modular fashion makes maintenance much easier.

Modules

- Decisions to be made during modular design:
  - What modules should a program have?
  - What services should each module provide?
  - How should the modules be interrelated?

Cohesion and Coupling

- In a well-designed program, modules should have two properties.
- ***High cohesion.*** The elements of each module should be closely related to one another.
  - High cohesion makes modules easier to use and makes the entire program easier to understand.
- ***Low coupling.*** Modules should be as independent of each other as possible.
  - Low coupling makes it easier to modify the program and reuse modules.

Interfaces of Modules: Header Files

Header Files

- Header files contains
  - Function declarations
  - Macro definitions to be shared between several source files
  - Type definitions to be shared between several source files

- By convention, header files have the extension `.h`.

Source Files

- A C program may contain any number of ***source files.***
- By convention, source files have the extension `.c`.
- Each source file contains part of the program, primarily definitions of functions and variables.
- One source file must contain a function named `main`, which serves as the starting point for the program.

Source Files

- A program containing multiple source files has significant advantages:
  - Grouping related functions and variables into a single file helps clarify the structure of the program.
  - Each source file can be compiled separately, which saves time.
  - Functions are more easily reused in other programs when grouped in separate source files.
  - With a header file, the related function declarations appear in only one place. No need to copy and past the functions (time-consuming and error-prone).

Header Files

- Problems that arise when a program contains several source files:
  - How can a function in one file call a function that's defined in another file?
  - How can two files share the same macro definition or type definition?

- The answer lies with the `#include` directive and header files, which makes it possible to share information among any number of source files.

Header Files

- The `#include` directive tells the preprocessor to insert the contents of a specified file.
- Information to be shared among several source files can be put into such a file.
- `#include` can then be used to bring the file's contents into each of the source files.
- Files that are included in this fashion are called ***header files*** (or sometimes ***include files***).

Header Files

- Header files contains
  - Function declarations
  - Macro definitions to be shared between several source files
  - Type definitions to be shared between several source files

- By convention, header files have the extension `.h`.

The `#include` Directive

- The `#include` directive has two primary forms.
- The first is used for header files that belong to C's own library:

  `#include` <*filename*>

- The second is used for all other header files:

  `#include "`*filename*`"`

- The difference between the two has to do with how the compiler locates the header file.

The `#include` Directive

- Typical rules for locating header files:
  - `#include` <*filename*>: Search the directory (or directories) in which system header files reside.

  - `#include` "*filename*": Search the current directory, then search the directory (or directories) in which system header files reside.

The `#include` Directive

- Don't use brackets when including header files that you have written:

```
#include <myheader.h>    /*** WRONG ***/
```

- The preprocessor will probably look for `myheader.h` where the system header files are kept.

The `#include` Directive

- The file name in an `#include` directive may include information that helps locate the file, such as a directory path or drive specifier:

```
#include "c:\cprogs\utils.h"
  /* Windows path */

#include "../include/utils.h"
  /* UNIX path */
```

- Although the quotation marks in the `#include` directive make file names look like string literals, the preprocessor doesn't treat them that way.

Sharing Macro Definitions and Type Definitions

- Most large programs contain macro definitions and type definitions that need to be shared by several source files.
- These definitions should go into header files.

- For example, the library `<limits.h>` header defines macros that represent the smallest and largest values of each integer type, for instance, `INT_MIN, INT_MAX`.

Sharing Macro Definitions and Type Definitions

- Suppose that a program uses macros named `TRUE`, and `FALSE` and typedef named `BOOL`,.
- Their definitions can be put in a header file with a name like `boolean.h`:

```
#define TRUE 1
#define FALSE 0
typedef int Bool;
```

- Any source file that requires these macros will simply contain the line

```
#include "boolean.h"
```

Sharing Macro Definitions and Type Definitions

- Advantages of putting definitions of macros and types in header files:
  - Makes the program easier to modify. Changing the definition of a macro or type requires editing a single header file.
  - Avoids inconsistencies caused by source files containing different definitions of the same macro or type.

Sharing Function Prototypes

- To reuse a function `f` defined in `foo.c`, a solution is to put `f`'s prototype in a header file (`foo.h`), then include the header file in all the places where `f` is called.

- We'll also need to include `foo.h` in `foo.c`, enabling the compiler to check that `f`'s prototype in `foo.h` matches its definition in `foo.c`.

Question

- Which one of the following should NOT be in a header file?

A) Macro definition

B) Type definition

C) Function definition

D) Function prototype

Program Design: Dividing a Program into Files

Dividing a Program into Files

- Designing a program involves determining what functions it will need and arranging the functions into logically related groups.
- Once a program has been designed, there is a simple technique for dividing it into files.

Dividing a Program into Files

- Each set of related functions will go into a separate source file (`foo.c`).

- Each source file will have a matching header file (`foo.h`).
  - `foo.h` will contain prototypes for the functions defined in `foo.c`.

Dividing a Program into Files

- `foo.h` will be included in each source file that needs to call a function defined in `foo.c`.

- **`foo.h` will also be included in `foo.c` so the compiler can check that the prototypes in `foo.h` match the definitions in `foo.c`.**

Dividing a Program into Files

- The `main` function will go in a file whose name matches the name of the program (the executable).
- It's possible that there are other functions in the same file as `main`, so long as they're not called from other files in the program.

```c
#define TRUE 1
#define FALSE 0
typedef int Bool;
//function prototypes
Bool logical_and(Bool a, Bool b);
Bool logical_or(Bool a, Bool b);
Bool logical_not(Bool a);
void print_bool(Bool b);
int main()
   {
          Bool a = TRUE;
          Bool b = FALSE;
          …
          print_bool(logical_and(a, b));
          …
          return 0;
   }
//function definitions
Bool logical_and(Bool a, Bool b){…}
Bool logical_or(Bool a, Bool b) {…}
Bool logical_not(Bool a) {…}
void print_bool(Bool b) {…}
```

Dividing a Program into Files

- For example, the `boolean.h` header file contains prototypes for functions:

```
#define TRUE 1
#define FALSE 0
typedef int Bool;
//function prototypes
Bool logical_and(Bool a, Bool b);
Bool logical_or(Bool a, Bool b);
Bool logical_not(Bool a);
void print_bool(Bool b);
```

Dividing a Program into Files

- The `boolean.c` file will contain definitions of the functions.

```
#include <stdio.h>
#include "boolean.h"
Bool logical_and(Bool a, Bool b)
{
    return (a&&b);
}
```

boolean.c Continued

```
Bool logical_or(Bool a, Bool b)
   {
         return (a||b);
   }
Bool logical_not(Bool a)
   {
         return (!a);
   }

void print_bool(Bool b)
   {
         printf("%s\n", (b ? "TRUE" : "FALSE"));
   }
```

Dividing a Program into Files

- Other source files might make use of the new type `Bool`, and the boolean functions. One example `booltest.c` that includes `"boolean.h"`:

```
#include <stdio.h>
#include "boolean.h"
int main()
{
    Bool a = TRUE;
    Bool b = FALSE;
    print_bool(logical_and(a, b));
    return 0;
}
```

Protecting Header Files

- If a source file includes the same header file twice, complication errors may result. This problem is common when header files include other header files.

- To avoid multiple inclusion, we'll enclose the contents of the header file in an `#ifndef-#endif` pair.

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;
...
#endif
```

Protecting Header Files: Conditional Compilation (Chapter 14, page 336)

- `#ifndef` tests whether an identifier is not defined as a macro, syntax:

```
#ifndef identifier
 …
#endif
```

- When the file was included the first time, the `BOOLEAN_H` macro won't be defined, so the preprocessor will allow the lines between `#ifndef` and `#endif` to stay.
- But if the file should be included a second time, the preprocessor will remove the lines between `#ifndef` and `#endif`

Programming Exercise

- Download `stack.c` in this week's in-class Exercises on Canvas.

- Divide the program into `stack.c` (containing functions that process the stack), `stack.h` (header file for `stack.c`), and `stack_test.c` (containing `main`)

- Compile and test the program:

```
gcc -o stack_test stack.c stack_test.c

./stack_test
```