Writing Large Programs (1)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Outline

- Source Files
- Header Files
- The `#include` Directive

Source Files

- Previously, we only write a C program that consists of a single file.
- In fact, a C program may be divided among any number of ***source files.***
- By convention, source files have the extension `.c`.

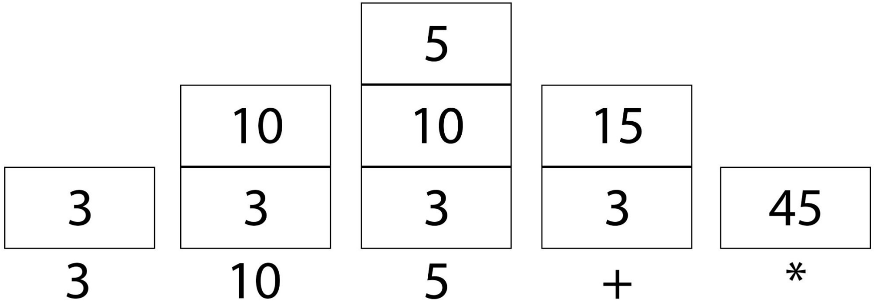
Source Files

- 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.
- Let’s use the following example to see how to write a program with multiple files

Source Files

- Consider the problem of writing a simple calculator program.
- The program will evaluate integer expressions entered in Reverse Polish notation (RPN), in which operators follow operands.

Equation: 3 10 5 + *



5

10 10 15

3 3 3 3 45

3 10 5 + *

slido

What is the result of 30 5 - 7 *?

ⓘ Start presenting to display the poll results on this slide.

slido

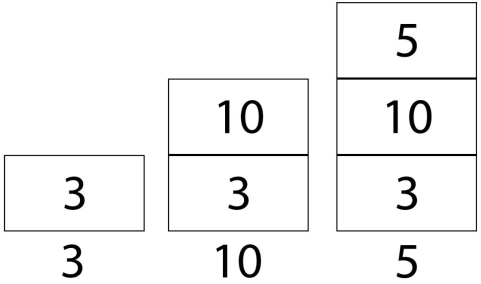
What kind of data structure is suitable to implement this program of RPN?

ⓘ Start presenting to display the poll results on this slide.

Source Files

- The program will read operands and operators, one by one, using a ____ to keep track of intermediate results.
  - If the program reads a number, it will push the number onto the stack.

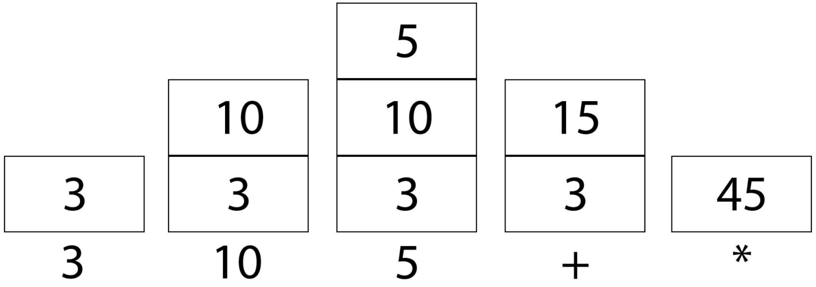
Equation: 3 10 5 + *



Source Files

- If the program reads an operator, it will pop two numbers from the stack, perform the operation, and then push the result back onto the stack.
- When the program reaches the end of the user’s input, the value of the expression will be on the stack.

Equation: 3 10 5 + *



Source Files

- How the expression `30 5 - 7 *`: will be evaluated
  - Push 30 onto the stack.
  - Push 5 onto the stack.
  - Pop the top two numbers from the stack, subtract 5 from 30, giving 25, and then push the result back onto the stack.
  - Push 7 onto the stack.
  - Pop the top two numbers from the stack, multiply them, and then push the result back onto the stack.
- The stack will now contain 175, the value of the expression.

Please draw the above figure of stack now to review and practice concept of stack

Source Files

- The program’s `main` function will contain a loop that performs the following actions:
  - Read a “token” (a number or an operator).
  - If the token is a number, push it onto the stack.
  - If the token is an operator, pop its operands from the stack, perform the operation, and then push the result back onto the stack.
- Turning the above strategy into a program is hard!

Source Files

- When dividing a program like this one into files, it makes sense to put related functions and variables into the same file.
- The function that reads tokens could go into one source file (`token.c`, say), together with any functions that have to do with tokens.
- Stack-related functions such as `push`, `pop`, `make_empty`, `is_empty`, and `is_full` could go into a different file, `stack.c`.
- The `main` function would go into yet another file, `calc.c`.

`token.c` `stack.c` `calc.c`

Source Files

- Splitting a program into 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.

Problems when dividing a program into several source files

- How can a function in one file call a function that’s defined in another file?
- How can a function access an external variable in another file?
- How can two files share the same macro definition or type definition?

?
token.c
stack.c
calc.c

Problems when dividing a program into several source files

- How can a function in one file call a function that's defined in another file?
- How can a function access an external variable in another file?
- How can two files share the same macro definition or type definition?

The answer lies with the `#include` directive, 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***).
- 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.
    - For example, on UNIX system, system header files are usually kept in the directory `/usr/include`
- `#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
- It’s usually best **not to** include path or drive information in #include directives.
- Why?

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

The `#include` Directive

- Such information make it difficult to compile a program whne it’s transported to another machine or another operating system!

The #include Directive

```
#include "d:utils.h"
#include "\cprogs\include\utils.h"
#include "d:\cprogs\include\utils.h"
```

```
#include "utils.h"
#include "..\include\utils.h"
```

The #include Directive

- The #include directive has a third form:

  #include *tokens*

- *tokens* is any sequence of preprocessing tokens.
- The preprocessor will scan the tokens and replace any **macros** that it finds.
- Let’s see an example directly!

The #include Directive

- After macro replacement, the resulting directive must match one of the other forms of `#include`.
- The advantage of the third kind of `#include` is that the file name can be defined by a macro rather than being “hard-coded” into the directive itself.

```
#if defined(IA32)
   #define CPU_FILE "ia32.h"
#elif defined(IA64)
   #define CPU_FILE "ia64.h"
#elif defined(AMD64)
   #define CPU_FILE "amd64.h"
#endif

#include CPU_FILE
```

Let’s Take A Break!

Sharing Macro Definitions and Type Definitions

- After talking about how to include header files, let’s talk about what should we put inside the header files!
- Most large programs contain
  - macro definitions
  - type definitions
  - function prototypes
- that need to be shared by several source files.
- These definitions should go into header files.

Sharing Macro Definitions and Type Definitions

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

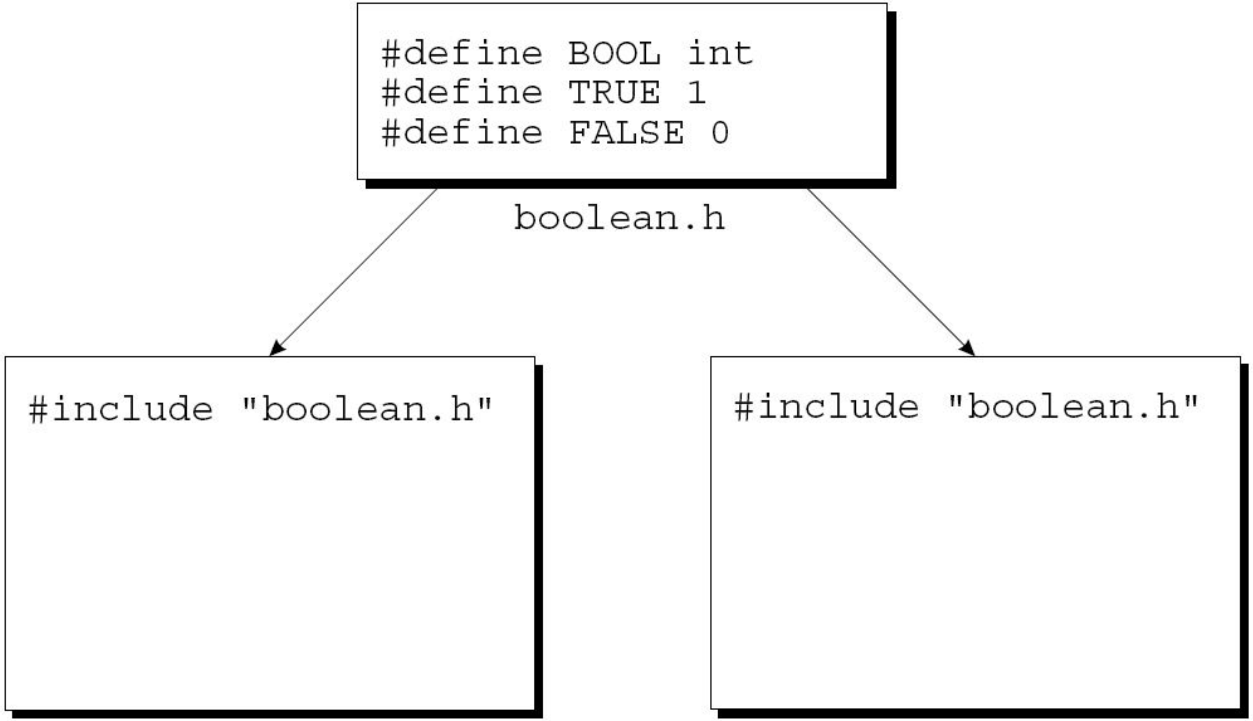
```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

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

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

Sharing Macro Definitions and Type Definitions

A program in which two files include `boolean.h`:



```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

boolean.h

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

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

Sharing Macro Definitions and Type Definitions

- Type definitions are also common in header files.
- For example, instead of defining a `BOOL` macro, we might use `typedef` to create a `Bool` type.
- If we do, the `boolean.h` file will have the following appearance:

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

Sharing Macro Definitions and Type Definitions

- Advantages of putting definitions of **macros** and **types** in header files:
  - **Saves time**. We don’t have to copy the definitions into the source files where they’re needed.
  - 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

- Suppose that a source file contains a call of a function `f` that’s defined in another file, `foo.c`.
- We already used these kinds of function a lot!
- For example, we use `printf()` which is defined in another file

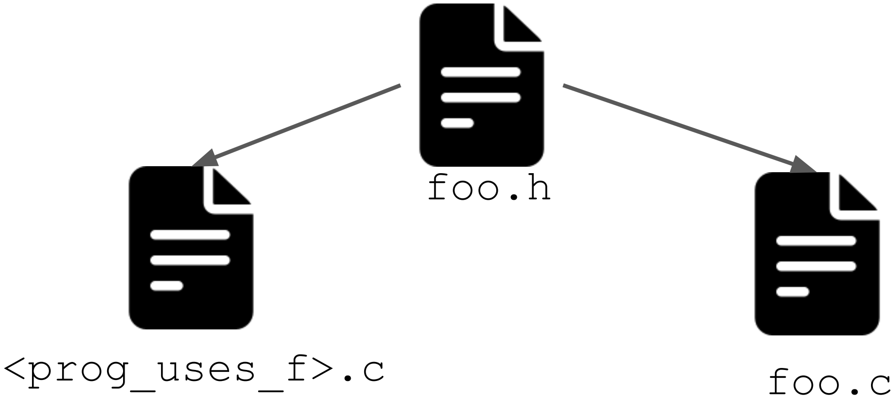
```
#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```

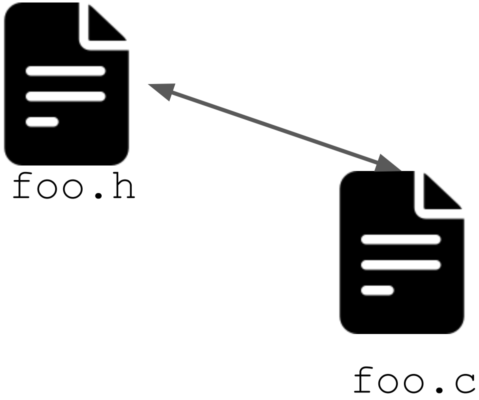
Sharing Function Prototypes

- To create our own functions like `printf()` that can be used by other files
- We need 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`.



Sharing Function Prototypes

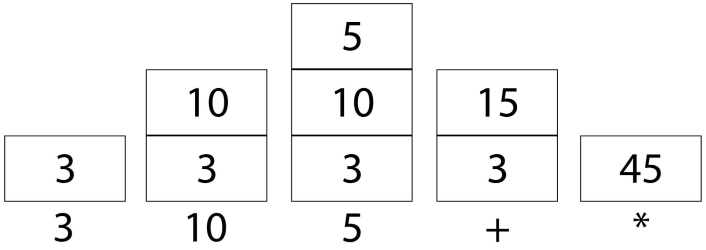
- If `foo.c` contains other functions, most of them should be declared in `foo.h`.
- Functions that are intended for use only within `foo.c` shouldn't be declared in a header file, however; to do so would be misleading.



Sharing Function Prototypes

- The Reverse Polish notation (RPN) calculator example can be used to illustrate the use of function prototypes in header files.

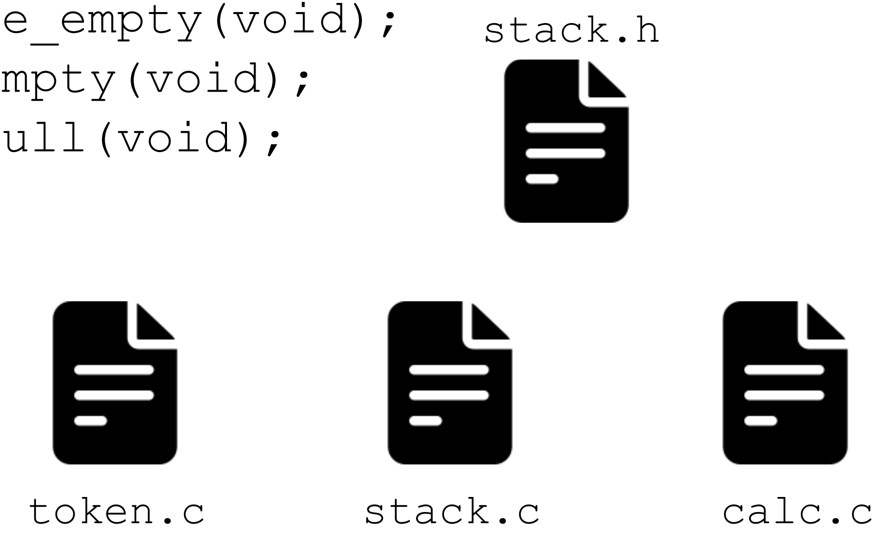
Equation: 3 10 5 + *



`token.c` `stack.c` `calc.c`

Sharing Function Prototypes

- The `stack.c` file will contain definitions of the `make_empty`, `is_empty`, `is_full`, `push`, and `pop` functions.
- Prototypes for these functions should go in the `stack.h` header file:
  - `void make_empty(void);`
  - `int is_empty(void);`
  - `int is_full(void);`
  - …



stack.h

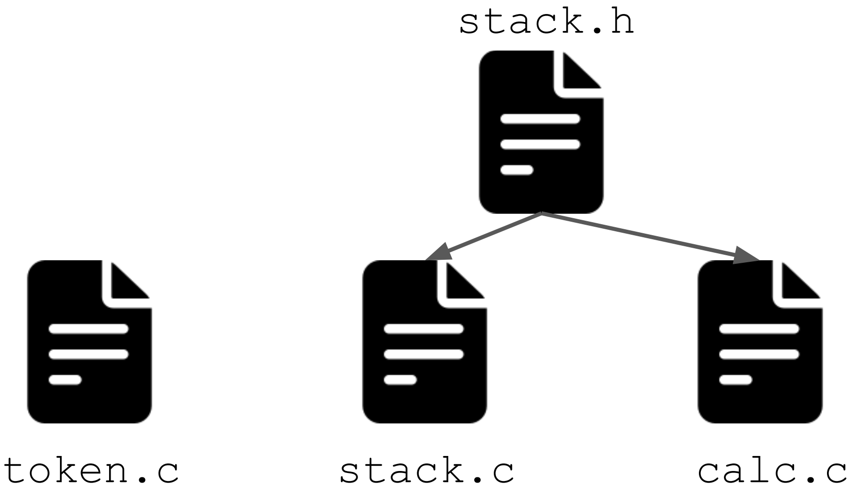
token.c

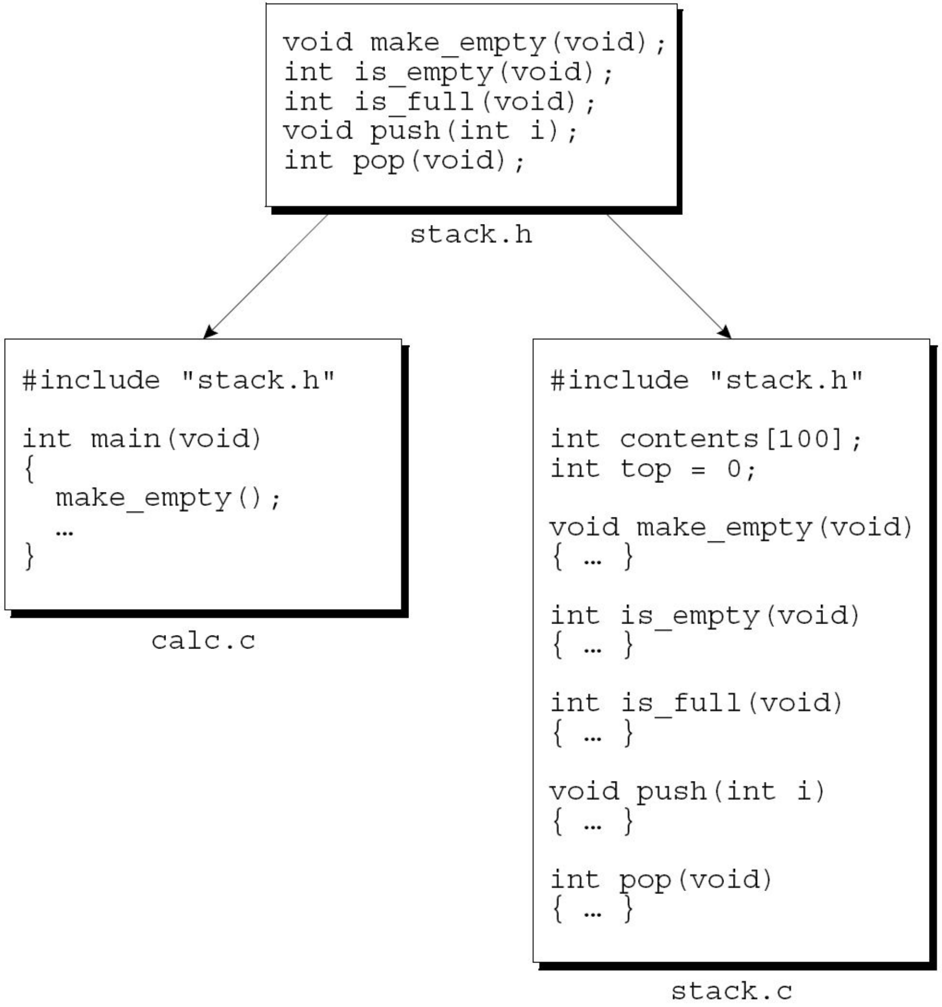
stack.c

calc.c

Sharing Function Prototypes

- We'll include `stack.h` in `calc.c` to allow the compiler to check any calls of stack functions that appear in the latter file.
- We'll also include `stack.h` in `stack.c` so the compiler can verify that the prototypes in `stack.h` match the definitions in `stack.c`.





```
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

stack.h

```
#include "stack.h"

int main(void)
{
  make_empty();
  …
}
```

calc.c

```
#include "stack.h"

int contents[100];
int top = 0;

void make_empty(void)
{ … }

int is_empty(void)
{ … }

int is_full(void)
{ … }

void push(int i)
{ … }

int pop(void)
{ … }
```

stack.c

Nested Includes

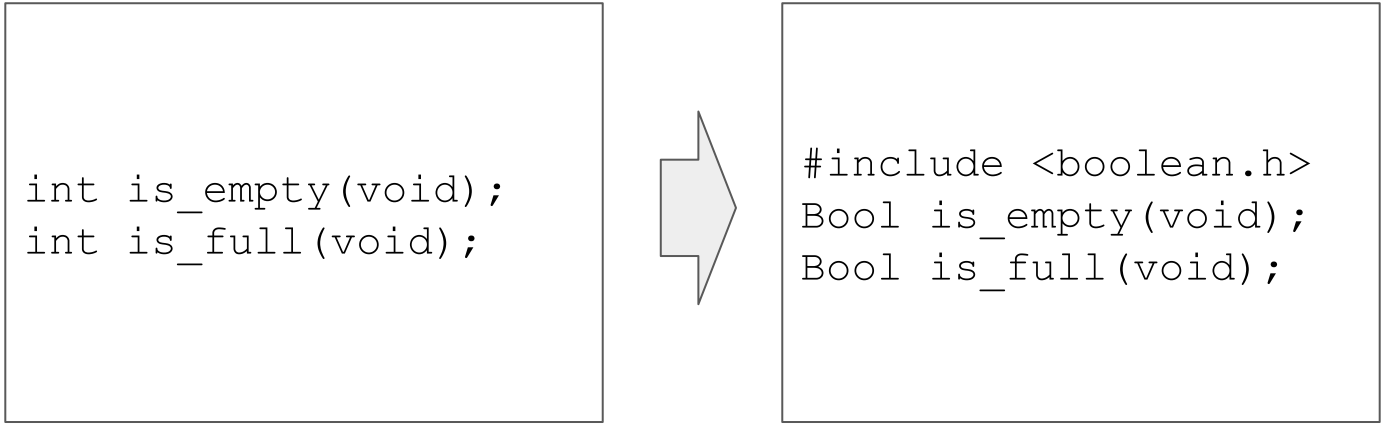
- A header file (`.h`) may contain `#include` directives.
- For example, `stack.h` contains the following prototypes:

```
int is_empty(void);
int is_full(void);
```

stack.h

Nested Includes

- Since these functions return only 0 or 1, it’s a good idea to declare their return type to be `Bool`
- We’ll need to include the `boolean.h` file in `stack.h` so that the definition of `Bool` is available when `stack.h` is compiled.



```
int is_empty(void);
int is_full(void);
```

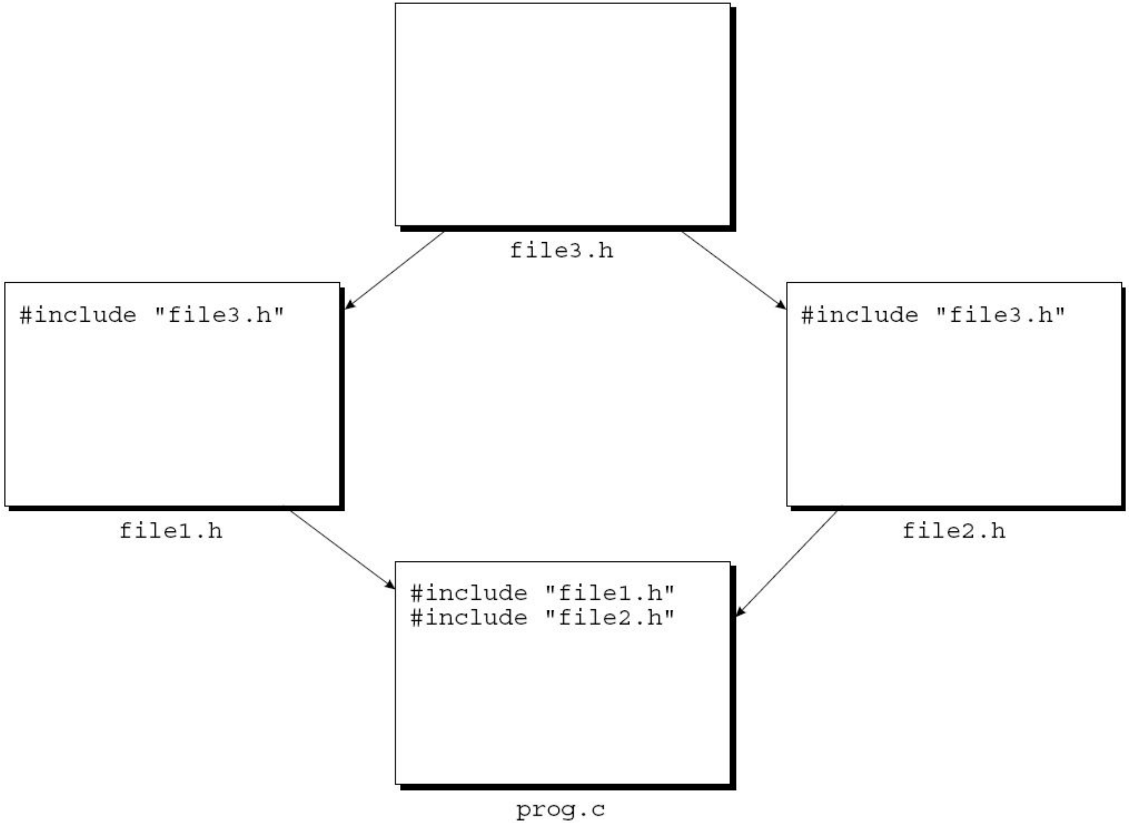
stack.h

```
#include <boolean.h>
Bool is_empty(void);
Bool is_full(void);
```

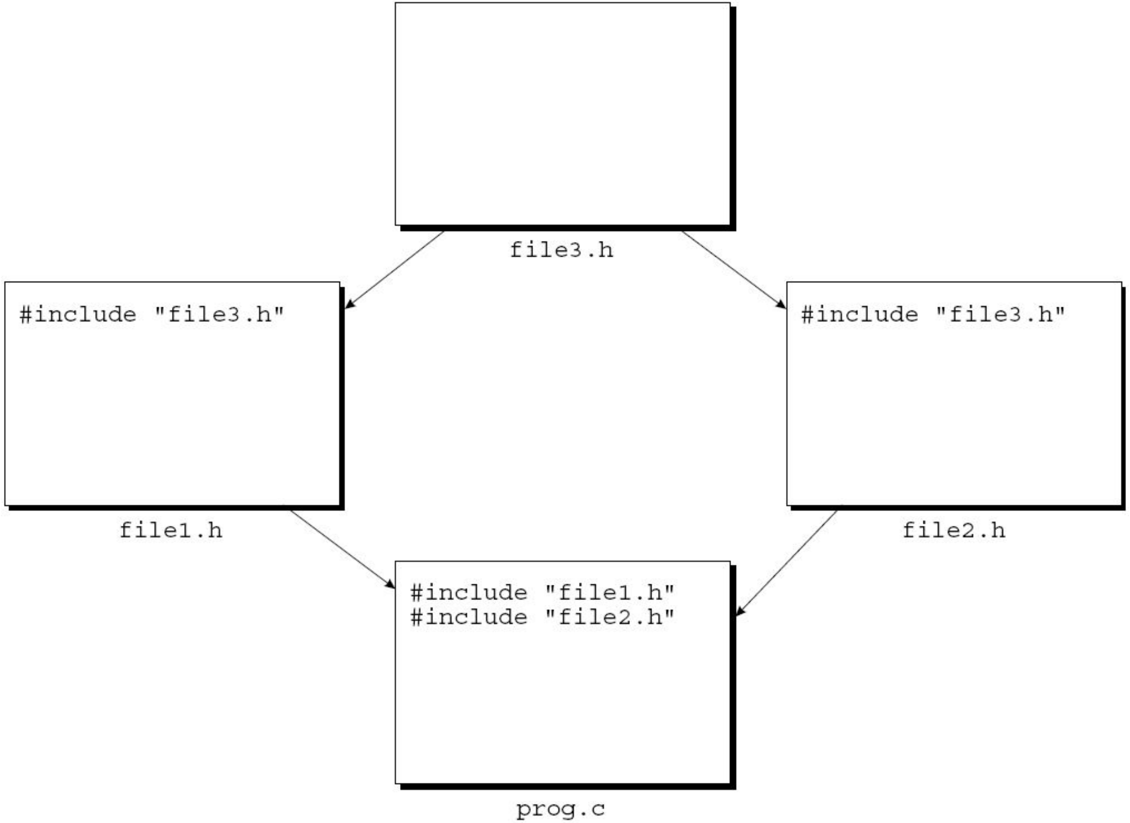
stack.h

Protecting Header Files

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



Suppose that `file1.h` includes `file3.h`, `file2.h` includes `file3.h`, and `prog.c` includes both `file1.h` and `file2.h`.



```
#include "file3.h"
```

```
#include "file3.h"
```

```
#include "file1.h"
#include "file2.h"
```

When `prog.c` is compiled, `file3.h` will be compiled twice.

Protecting Header Files

- To be safe, it’s probably a good idea to **protect all header files** against **multiple** inclusion.
- In addition, we might **save some time** during program development by avoiding unnecessary recompilation of the same header file.

Protecting Header Files

- To protect a header file, we’ll enclose the contents of the file in an `#ifndef`-`#endif` pair.
- While this header file is included in the firs time, the `BOOLEAN_H` macro won’t be defined
- So preprocessor will allow the lines between `#ifndef` and `#endif` to stay

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

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

#endif
```

Protecting Header Files

- But, if this header file is included a second time, the preprocessor will remove the line between `#ifndef` and `#endif`

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

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

#endif
```

Protecting Header Files

- The name BOOLEAN_H doesn’t really matter.
- But, Making name of the macro resemble the name of the header file is a good way to avoid conflicts with other macros.
- Since we can’t name the macro BOOLEAN.H, a name such as BOOLEAN_H is a good alternative.

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

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

#endif
```

Summary

- The `#include` Directive
    - three forms
- Sharing Macro Definitions and Type Definitions
- Sharing Function Prototypes
- Nested Includes
- Protecting Header Files