Modules

CS2023 Winter 2004

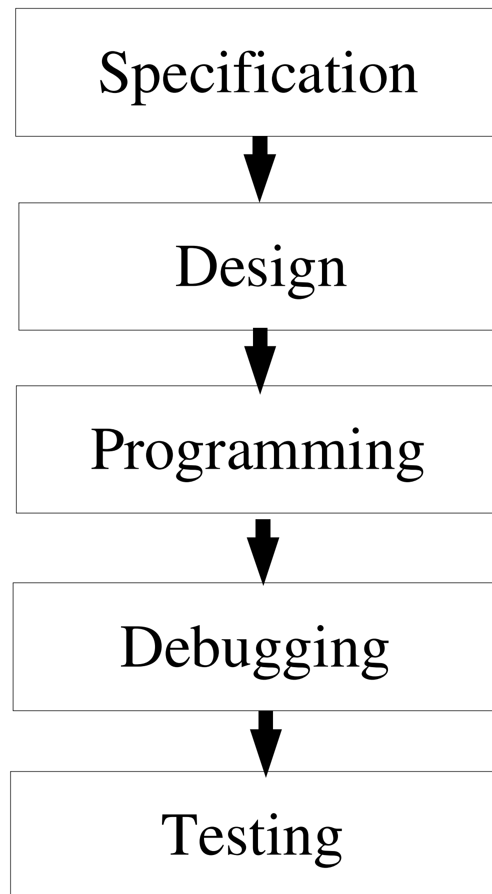
Outcomes: Modules

- "C for Java Programmers", Chapter 7, sections 7.4.1 - 7.4.6
- Code Complete, Chapter 6
- After the conclusion of this section you should be able to
  - Understand why modules are needed
  - Start creating your own modules, complete with header files
  - Understand the relationship between client, interface, and implementation

Functional vs Modular Decomposition

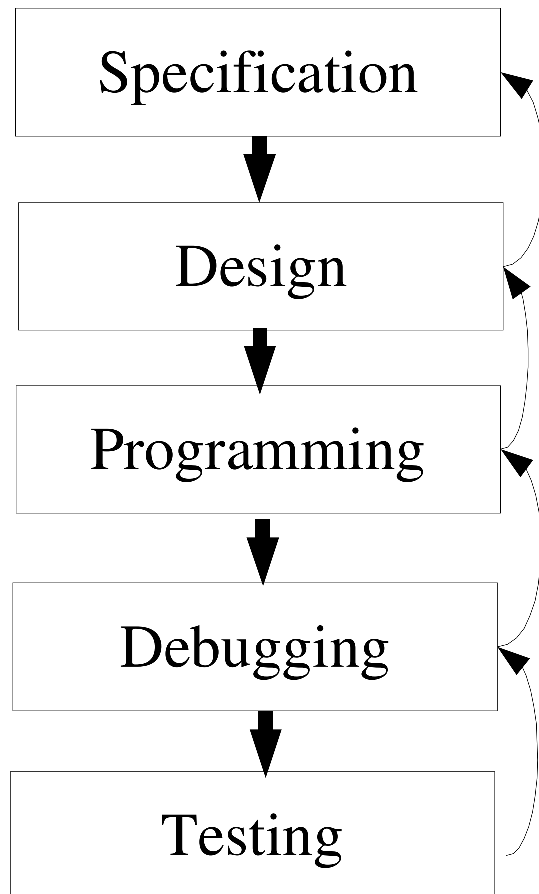
- Goal of functional (procedural) decomposition:
  - Collection of loosely coupled and strongly cohesive functions
  - In reality many functions will be coupled through the data they share

Software in CS2023



1 Person
10^2 Lines of Code
1 Type of Machine
0 Modifications
1 Week

Software in the Real World



Lots of People
10^6 Lines of Code
Lots of Machines
Lots of Modifications
1 Decade or more

Good Software in the Real World

- Understandable
  - Well-designed
  - Consistent
  - Documented
- Robust
  - Works for any input
  - Tested
- Reusable
  - Components
- Efficient
  - Only matters for 1%

Good Software in the Real World

- Understandable
  - Well-designed
  - Consistent
  - Documented
- Robust
  - Works for any input
  - Tested
- Reusable
  - Components
- Efficient
  - Only matters for 1%

Write code in modules
with well-defined interfaces

Write code in modules
and test them separately

Write code in modules
that can be used elsewhere

Write code in modules
and optimize the slow ones

Functional vs Modular Decomposition

- Goal of modular decomposition:
  - “Allow one module to be written with little knowledge of the code in another module
  - Allow modules to be reassembled and replaced without reassembly of the whole system”

    David Parnas, “On the Criteria To Be Used in Decomposing Systems into Modules”, Comm. ACM vol. 15 no. 12, 1053 (1972)

Modules

- Programs are made up of many modules
- Each module is small and does one thing
  - string manipulation
  - mathematical functions
  - Set, stack, queue, list, ...
- Deciding how to break up a program into modules is a key to good software design

Modularity: Cohesion & Coupling

- Want each module to be a black box with a well-defined functionality
- Module cohesion
  - Cohesive data and functionality
  - Group of services that belong together
  - E.g. cruise control simulator:
    - Set the speed
    - Resume former speed
    - Deactivate
    - Plus functions for internal use

Modularity: Cohesion & Coupling

- Module Coupling
  - Offer complete set of services (through interface)
  - *GetCurrentSettings(), SetSpeed(), ResumeFormerSpeed(), Deactivate()*
  - If services incomplete, other functions might have to read or write its internal data directly
  - Functions in a module will be strongly coupled through its data
    - Not as bad as global data, which is visible to all functions
    - Modular data can be kept hidden from other modules

Modules: RPN Calculator

- Evaluate integer expressions entered in Reverse Polish notation
  - *e.g.* user enters expression: **`30 5 - 7 *`**
  - program prints its value: **`175`**
- Program reads numbers and operators one by one
  - number: push it onto stack
  - operator:
    - pop two numbers
    - perform operation
    - push result

Modules: RPN Calculator

- Problem decomposition:

1. Stack representation and operations
    - **`stack.c`**
        - Push, Pop, *etc.*.
2. Main program
    - **`calc.c`**

Modules: RPN Calculator

- Main program:

while more tokens

read a token

if token is a number

push it on a stack

if token is an operator

pop its operands from stack

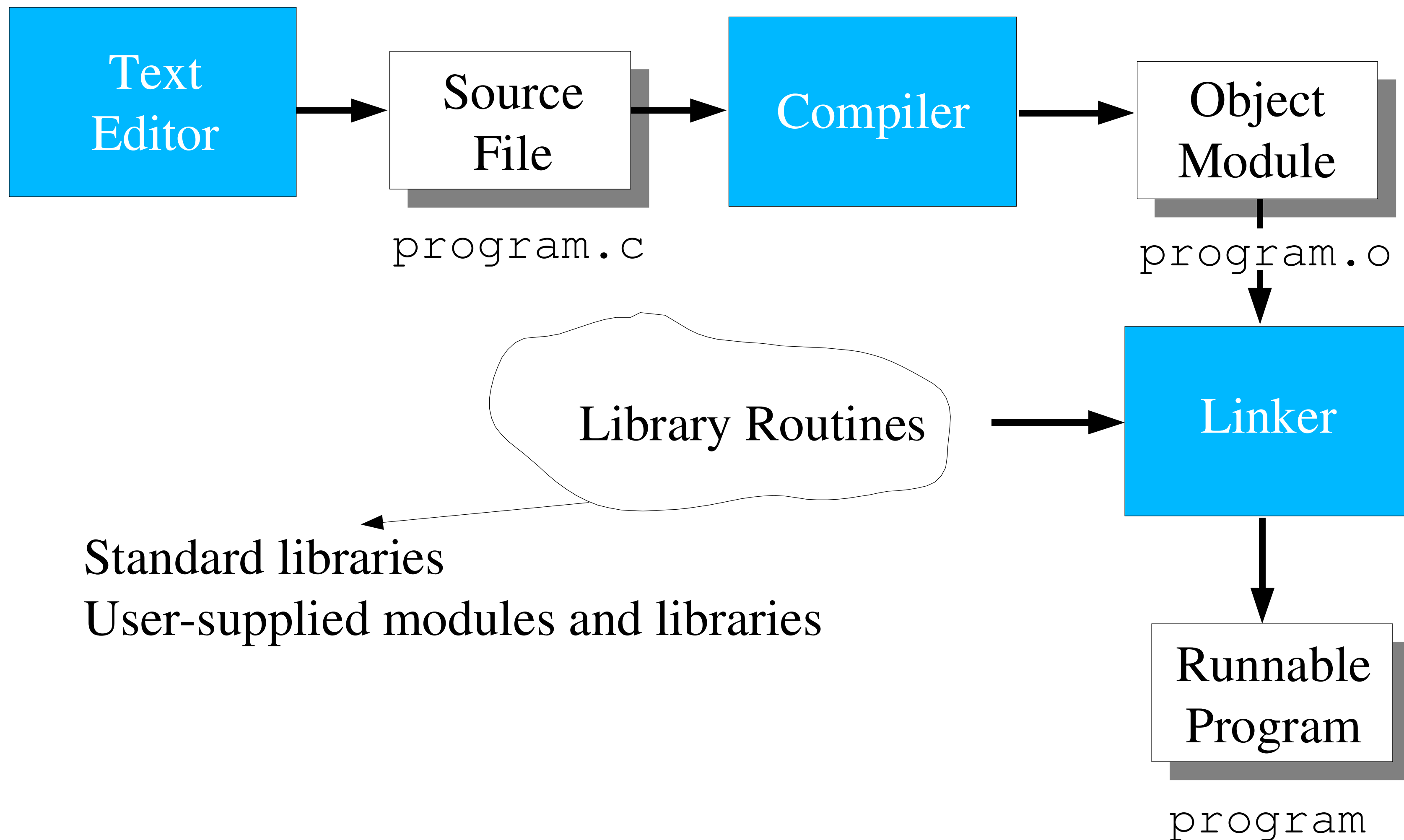
perform operation

push result back in stack

Modules in Separate Files

- Each module can be compiled separately
  - saves time!
  - **`gcc -c stack.c`**
    - compiles but does link
    - produces **`stack.o`** object code
- Modules more easily reused if kept in separate files
- How can a function in one file call a function that's defined in another file?

Recall: from source file to runnable program



Building multi-file programs

- Compile modules separately, to produce object files for each module (such as `stack.o`)
- When compiling main program, give also name of module object files on command line

```
gcc -o calc calc.c stack.o
```

  - Can also compile and link all files at same time:

```
gcc -o calc calc.c stack.c
```

Header Files

- A function may be defined in one file and called in another file, as long as the call is preceded by the function declaration.
- Recall:
  - include header files for standard C library:

```
#include <filename>
```

Search directory(ies) in which system header files reside (usually /usr/include)

Header Files

– include all other header files (including programmer's own)

#include "*filename*"

Search current directory

Best not to include absolute path:

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

#include "utils.h"
#include "../include/utils.h"

Header Files: Macros and Typedef

- Most large programs contain macro and type definitions that need to be shared by several source files
- *e.g.* need macros **`TRUE`**, **`FALSE`**, and **`Bool`** type
- Instead of repeating definitions in each source file, put them in a header file!
  - save time copying definitions
  - program easier to modify

Boolean Header File

boolean.h:

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

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

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

Headers and Function Prototypes

- Already seen that calling functions without declaring them is risky!
- Why not declare function in file where it's called?
- Better to put declaration in a header file and include header file in places where function is called
  - if function **`f`** is defined in **`foo.c`**, put its declaration in **`foo.h`**
  - include **`foo.h`** in files where **`f`** is called
  - do we need to include **`foo.h`** in **`foo.c`**?

Headers and Function Prototypes

square.c:

```
double square(double x)
{
   return x * x;
}
```

square.h:

```
int square(double x);
```

square_test.c:

```
#include <stdio.h>
#include "square.h"

int main()
{

int i;
scanf("%d", &i);
printf("%g\n",
  square(i))

return 0;
}
```

Headers and Function Prototypes

- Always include the header file declaring a function **`f`** in the source file that contains **`f`**'s definition.
- If **`foo.c`** contains other functions, most should be declared in same header file as **f**.
- Functions that are only needed by **`foo.c`** shouldn't be declared in **`foo.h`**
  - "secrets" user of module doesn't need to know

RPN Program

Stack.h

```
void MakeEmpty();
int IsEmpty();
int IsFull();
void Push(int i);
int Pop();
```

stack.c

```
#include "stack.h"

/* global var. */

void MakeEmpty()
{...}
int IsEmpty()
{...}
int IsFull()
{...}
void Push(int i)
{...}
int Pop()
{...}
```

calc.c

```
#include "stack.h"

main()
{
...
  MakeEmpty();
...
}
```

Sharing Variables

- To share a function, put its *definition* in one source file, and *declarations* in files that need to call the function
- Sharing variables done in much the same way
  - **`int i;`**
  - **`i`** declared to be a variable of type **`int`**, and defined by compiler setting aside space for **`i`**
- To declare **`i`** without defining it, use **`extern`** keyword:
  - **`extern int i;`**
- extern informs compiler that **`i`** is defined elsewhere in the program, so no need to allocate space for it.

Sharing Variables

- When declarations of the same variable appear in different files, compiler can't check that declarations match variable's definition
  - similar to problem with functions (see earlier)
- One file: **`int i;`**
- Another file: **`extern double i;`**
- Declarations of shared variables usually put in header files
- Sharing variables defeats the purpose of modularization

Nested Includes

- A header file may itself contain **`#include`** directives
- **`stack.h:`**

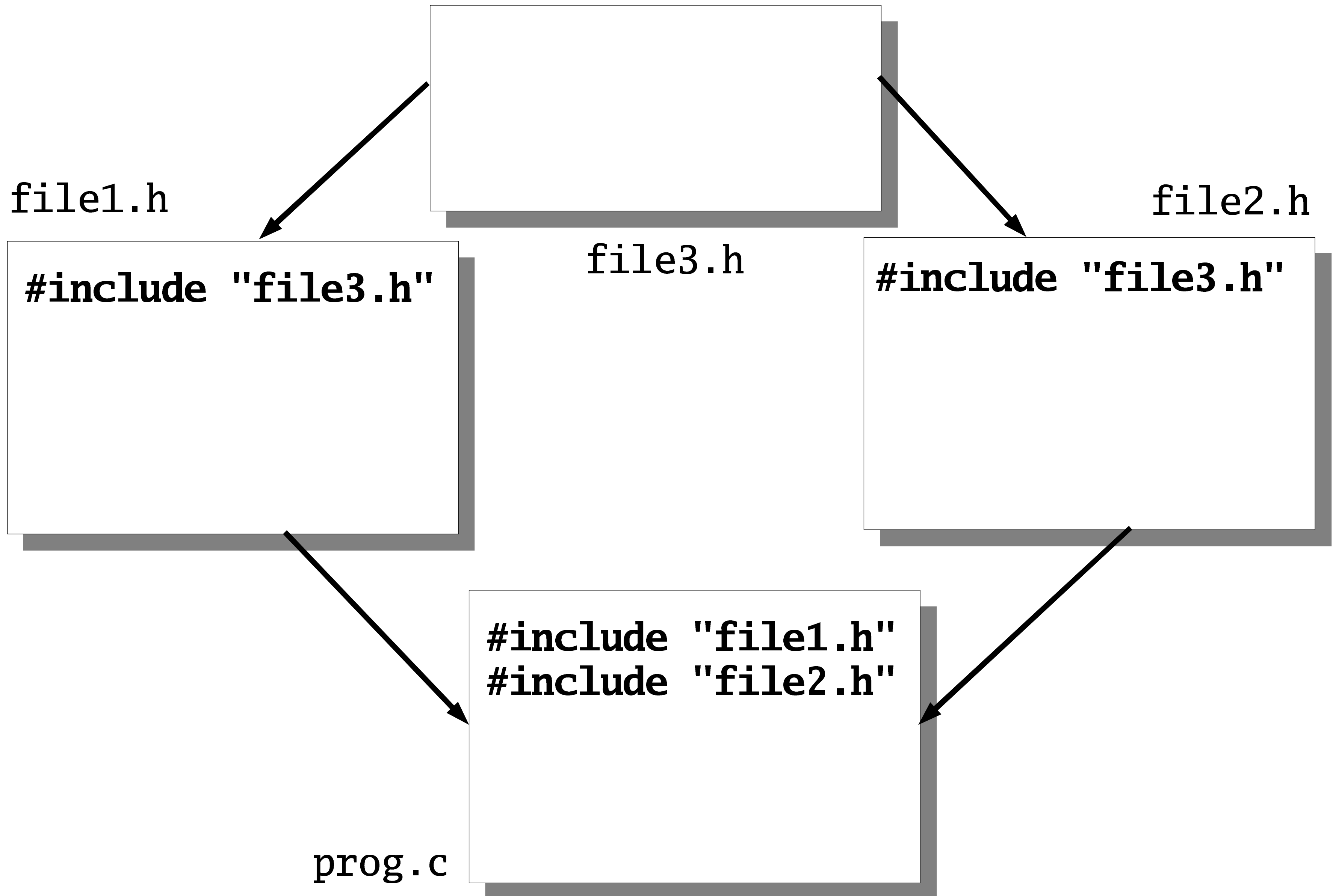
```
int IsEmpty();
int IsFull();
```

- Use **`Bool`** type:

```
Bool IsEmpty();
Bool IsFull();
```

- Need to include **`boolean.h`** in **`stack.h`**

"file3.h" included twice



Protecting Header Files

- Multiple inclusion not a concern if header file contains only macro definitions, function prototypes, or variable declarations
- If header file contains type definition, compilation error will result
- **`boolean.h:`**

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

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

#endif
```

Interfaces

- An interface defines what the module does
  - decouples clients from implementation
  - hide implementation details
- An interface specifies
  - data types and variables
  - Functions that may be invoked
- Interfaces are defined in header files

stack.h

```
void MakeEmpty();
int IsEmpty();
int IsFull();
void Push(int i);
int Pop();
```

Implementations

- An implementation defines how the module does it
- Can have many implementations for one interface
  - different algorithms for different situations
  - Machine dependencies, efficiency, etc...
- Implementations are defined in source files (.c)

stack.c

```
int IsEmpty()
{
....
}
```

Clients

- A client uses a module via its interface
- Clients see only the interface
  - can use module without knowing its implementation
- Client is unaffected if implementation changes
  - As long as interface stays the same
- Clients "include" header files

```
include "stack.h"
int main(){
  while(scanf("%d", &i)!=EOF)
    push(i)
....
}
```

Clients, Interfaces, Implementations

- Interfaces are contracts between clients and implementations
  - Clients must use interface correctly
  - Implementations must do what they advertise