

Scope, Blocks, and Modularity

Goals of this Lecture

- Help you learn:
 - Local vs. global variables, scope, and blocks
 - How to create high quality modules in C
- Why?
 - Knowing lifetime and visibility of identifiers is crucial in writing correct code
 - Abstraction is a powerful (the only?) technique available for understanding large, complex systems
 - A power programmer knows how to find the abstractions in a large program
 - A power programmer knows how to convey a large program's abstractions via its modularity

Local Variables

- A variable declared in the body of a function is said to be ***local*** to the function:

```
int sum_digits(int n)
{
    int sum = 0;    /* local variable */

    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }

    return sum;
}
```


Local Variables

- Default properties of local variables:
 - ***Automatic storage duration.*** Storage is “automatically” allocated when the enclosing function is called and deallocated when the function returns.
 - ***Block scope.*** A local variable is **visible** from its point of declaration to the end of the enclosing function body.

Local Variables

- Since C99 doesn't require variable declarations to come at the beginning of a function, it's possible for a local variable to have a very small scope:

```
void f(void)
{
    ...
    int i;
    ...
}
```



scope of i

Static Local Variables

- Including `static` in the declaration of a local variable causes it to have ***static storage duration***.
- A variable with static storage duration has a permanent storage location, so it retains its value throughout the execution of the program.

- Example:

```
void f(void)
{
    static int i;    /* static local variable */
    ...
}
```

- A static local variable still has block scope, so it's not visible to other functions.

Function Parameters

- Parameters have the same properties—automatic storage duration and block scope—as local variables.
- Each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

External Variables

- Passing arguments is one way to transmit information to a function.
- Functions can also communicate through ***external variables***—variables that are declared outside the body of any function.
- External variables are sometimes known as ***global variables***.

External Variables

- Properties of external variables:
 - Static storage duration
 - File scope
- Having ***file scope*** means that an external variable is visible from its point of declaration to the end of the enclosing file.

Example: Using External Variables to Implement a Stack

- To illustrate how external variables might be used, let's look at a data structure known as a ***stack***.
- A stack, like an array, can store multiple data items of the same type.
- The operations on a stack are limited:
 - ***Push*** an item (add it to one end—the “stack top”)
 - ***Pop*** an item (remove it from the same end)
- Examining or modifying an item that's not at the top of the stack is forbidden.

Example: Using External Variables to Implement a Stack

- One way to implement a stack in C is to store its items in an array, which we'll call `contents`.
- A separate integer variable named `top` marks the position of the stack top.
 - When the stack is empty, `top` has the value 0.
- To *push* an item: Store it in `contents` at the position indicated by `top`, then increment `top`.
- To *pop* an item: Decrement `top`, then use it as an index into `contents` to fetch the item that's being popped.

Example: Using External Variables to Implement a Stack

- The following program fragment declares the `contents` and `top` variables for a stack.
- It also provides a set of functions that represent stack operations.
- All five functions need access to the `top` variable, and two functions need access to `contents`, so `contents` and `top` will be external.

Example: Using External Variables to Implement a Stack

```
#include <stdbool.h>    /* C99 only */

#define STACK_SIZE 100

/* external variables */
int contents[STACK_SIZE];
int top = 0;

void make_empty(void)
{
    top = 0;
}

bool is_empty(void)
{
    return top == 0;
}
```

Example: Using External Variables to Implement a Stack

```
bool is_full(void)
{
    return top == STACK_SIZE;
}

void push(int i)
{
    if (is_full())
        stack_overflow();
    else
        contents[top++] = i;
}

int pop(void)
{
    if (is_empty())
        stack_underflow();
    else
        return contents[--top];
}
```

Pros and Cons of External Variables

- External variables are convenient when many functions must share a variable or when a few functions share a large number of variables.
- In most cases, it's better for functions to communicate through parameters rather than by sharing variables:
 - If we change an external variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.
 - If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function.
 - Functions that rely on external variables are hard to reuse in other programs.

Pros and Cons of External Variables

- Making variables external when they should be local can lead to some rather frustrating bugs.
- Code that is supposed to display a 10 × 10 arrangement of asterisks:

```
int i;

void print_one_row(void)
{
    for (i = 1; i <= 10; i++)
        printf("*");
}

void print_all_rows(void)
{
    for (i = 1; i <= 10; i++) {
        print_one_row();
        printf("\n");
    }
}
```

- Instead of printing 10 rows, `print_all_rows` prints only one.

Blocks

- We encountered compound statements of the form:

{ *statements* }

- C allows compound statements to contain declarations as well as statements:

{ *declarations statements* }

- This kind of compound statement is called a ***block***.

Blocks

- Example of a block:

```
if (i > j) {  
    /* swap values of i and j */  
    int temp = i;  
    i = j;  
    j = temp;  
}
```

Blocks

- By default, the storage duration of a variable declared in a block is **automatic**: storage for the variable is allocated when the block is entered and deallocated when the block is exited.
- The variable has block scope; it can't be referenced outside the block.
- A variable that belongs to a block can be declared `static` to give it static storage duration.

Blocks

- The body of a function is a block.
- Blocks are also useful inside a function body when we need variables for temporary use.
- Advantages of declaring temporary variables in blocks:
 - Avoids cluttering declarations at the beginning of the function body with variables that are used only briefly.
 - Reduces name conflicts.
- C99 allows variables to be declared anywhere within a block.

Scope

- Scope defines the visible area of a given identifier
- C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.
- The most important scope rule: When a declaration inside a block names an identifier that's already visible, the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning.
- At the end of the block, the identifier regains its old meaning.

```

int i ;           /* Declaration 1 */

void f(int i )   /* Declaration 2 */
{
    i = 1;
}

void g(void)
{
    int i = 2;    /* Declaration 3 */
    if (i > 0) {
        int i ;  /* Declaration 4 */
        i = 3;
    }
    i = 4;
}

void h(void)
{
    i = 5;
}

```

Modularity

- Good program consists of well-designed modules (set of code that provides related functionalities)
- Let's learn how to design a good module

Interfaces

- (1) A well-designed module separates interface and implementation
- Why?
 - Hides implementation details from clients
 - Thus facilitating abstraction
 - Allows separate compilation of each implementation
 - Thus allowing partial builds

Interface Example

- Stack: A stack whose items are strings
 - Data structure
 - Linked list
 - Algorithms
 - **new**: Create a new Stack object and return it (or NULL if not enough memory)
 - **free**: Free the given Stack object
 - **push**: Push the given string onto the given Stack object and return 1 (or 0 if not enough memory)
 - **top**: Return the top item of the given Stack object
 - **pop**: Pop a string from the given Stack object and discard it
 - **isEmpty**: Return 1 the given Stack object is empty, 0 otherwise

Interfaces Example

- Stack (version 1)

```
/* stack.c */
```

```
struct Node {  
    const char *item;  
    struct Node *next;  
};  
  
struct Stack {  
    struct Node *first;  
};
```

```
struct Stack *Stack_new(void) {...}  
void Stack_free(struct Stack *s) {...}  
int Stack_push(struct Stack *s, const char *item) {...}  
char *Stack_top(struct Stack *s) {...}  
void Stack_pop(struct Stack *s) {...}  
int Stack_isEmpty(struct Stack *s) {...}
```

```
/* client.c */
```

```
#include "stack.c"
```

```
/* Use the functions  
defined in stack.c. */
```

- Stack module consists of one file (stack.c); no separate interface
- Problem: Change stack.c => must rebuild stack.c **and client**
- Problem: Client "sees" Stack function definitions; poor abstraction

Interfaces Example

- Stack (version 2)

```
/* stack.h */

struct Node {
    const char *item;
    struct Node *next;
};
struct Stack {
    struct Node *first;
};

struct Stack *Stack_new(void);
void Stack_free(struct Stack *s);
int Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void Stack_pop(struct Stack *s);
int Stack_isEmpty(struct Stack *s);
```

- Stack module consists of two files:
 - (1) stack.h (the interface) declares functions and defines data structures

Interfaces Example

- Stack (version 2)

```
/* stack.c */
#include "stack.h"

struct Stack *Stack_new(void) {...}
void Stack_free(struct Stack *s) {...}
int Stack_push(struct Stack *s, const char *item) {...}
char *Stack_top(struct Stack *s) {...}
void Stack_pop(struct Stack *s) {...}
int Stack_isEmpty(struct Stack *s) {...}
```

(2) stack.c (the implementation) defines functions

- #includes stack.h so
 - Compiler can check consistency of function declarations and definitions
 - Functions have access to data structures

Interfaces Example

- Stack (version 2)

```
/* client.c */  
  
#include "stack.h"  
  
/* Use the functions declared in stack.h. */
```

- Client #includes only the interface
- Change stack.c => must rebuild stack.c, ***but not the client***
- Client does not “see” Stack function definitions; better abstraction

Encapsulation

- (2) A well-designed module encapsulates data
 - An interface should hide implementation details
 - A module should use its functions to encapsulate its data
 - A module should not allow clients to manipulate the data directly
- Why?
 - **Clarity**: Encourages abstraction
 - **Security**: Clients cannot corrupt object by changing its data in unintended ways
 - **Flexibility**: Allows implementation to change – even the data structure – without affecting clients

Encapsulation Example

- Stack (version 1)

```
/* stack.h */  
  
struct Node {  
    const char *item;  
    struct Node *next;  
};  
struct Stack {  
    struct Node *first;  
};  
  
struct Stack *Stack_new(void);  
void Stack_free(struct Stack *s);  
void Stack_push(struct Stack *s, const char *item);  
char *Stack_top(struct Stack *s);  
void Stack_pop(struct Stack *s);  
int Stack_isEmpty(struct Stack *s);
```

Structure type definitions
in .h file

- That's bad
- Interface reveals how Stack object is implemented (e.g., as a linked list)
- Client can access/change data directly; could corrupt object

Encapsulation Example

- Stack (version 2)

```
/* stack.h */  
  
struct Stack;  
  
struct Stack *Stack_new(void);  
void Stack_free(struct Stack *s);  
void Stack_push(struct Stack *s, const char *item);  
char *Stack_top(struct Stack *s);  
void Stack_pop(struct Stack *s);  
int Stack_isEmpty(struct Stack *s);
```

Move definition of struct Node to implementation; clients need not know about it

Place **declaration** of struct Stack in interface; move **definition** to implementation

- That's better
- Interface does not reveal how Stack object is implemented
- Client cannot access data directly

Encapsulation Example 1

- Stack (version 3)

```
/* stack.h */  
  
typedef struct Stack * Stack_T;  
  
Stack_T Stack_new(void);  
void Stack_free(Stack_T s);  
void Stack_push(Stack_T s, const char *item);  
char *Stack_top(Stack_T s);  
void Stack_pop(Stack_T s);  
int Stack_isEmpty(Stack_T s);
```

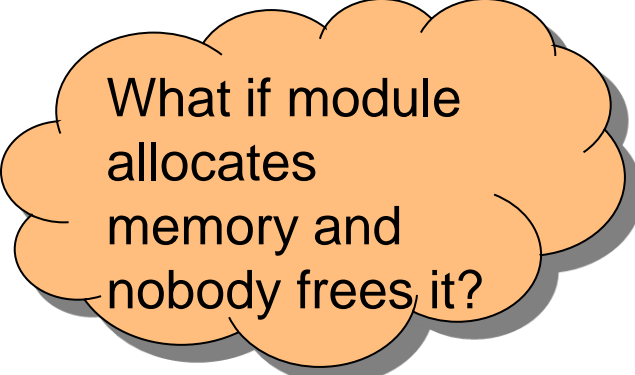
Opaque pointer

- That's better still
- Interface provides "Stack_T" abbreviation for client
- Interface encourages client to view a Stack as an object, not as a (pointer to a) structure
- Client still cannot access data directly; data is "opaque" to the client

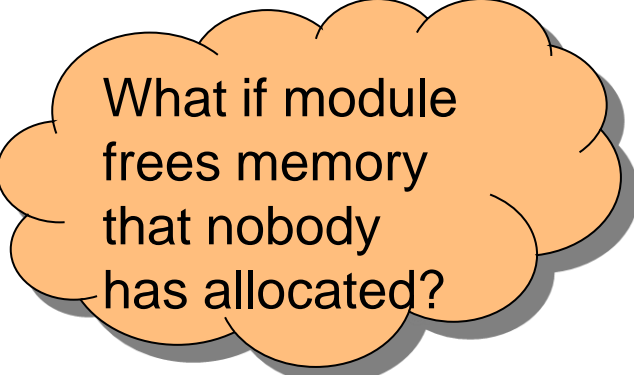
Resources

(3) A well-designed module manages resources consistently

- A module should free a resource if and only if the module has allocated that resource
- Examples
 - Object allocates memory \Leftrightarrow object frees memory
 - Object opens file \Leftrightarrow object closes file
- Why?
 - Error-prone to allocate and free resources at different levels



What if module
allocates
memory and
nobody frees it?



What if module
frees memory
that nobody
has allocated?

Resources Example

- Stack: Who allocates and frees the strings?
 - Reasonable options:
 - (1) Client allocates and frees strings
 - `Stack_push()` does not create copy of given string
 - `Stack_pop()` does not free the popped string
 - `Stack_free()` does not free remaining strings
 - (2) Stack object allocates and frees strings
 - `Stack_push()` creates copy of given string
 - `Stack_pop()` frees the popped string
 - `Stack_free()` frees all remaining strings
 - Our choice: (1)



Advantages/
disadvantages?

SymTable Aside

- Consider SymTable (from Assignment 3)...
- Who allocates and frees the key strings?
 - Reasonable options:
 - (1) Client allocates and frees strings
 - `SymTable_put()` does not create copy of given string
 - `SymTable_remove()` does not free the string
 - `SymTable_free()` does not free remaining strings
 - (2) SymTable object allocates and frees strings
 - `SymTable_put()` creates copy of given string
 - `SymTable_remove()` frees the string
 - `SymTable_free()` frees all remaining strings
 - Our choice: (2)



Advantages/
disadvantages?

Passing Resource Ownership

- Passing resource ownership
 - Should note violations of the heuristic in function comments

```
/* somefile.h */  
  
...  
  
void *f(void);  
/* ...  
    This function allocates memory for  
    the returned object.  You (the caller)  
    own that memory, and so are responsible  
    for freeing it when you no longer  
    need it. */  
  
...
```

Consistency

- (4) **A well-designed module is consistent**
- A function's name should indicate its module
 - Facilitates maintenance programming; programmer can find functions more quickly
 - Reduces likelihood of name collisions (from different programmers, different software vendors, etc.)
 - A module's functions should use a consistent parameter order
 - Facilitates writing client code

Consistency Examples

- Stack
 - (+) Each function name begins with "Stack_"
 - (+) First parameter identifies Stack object

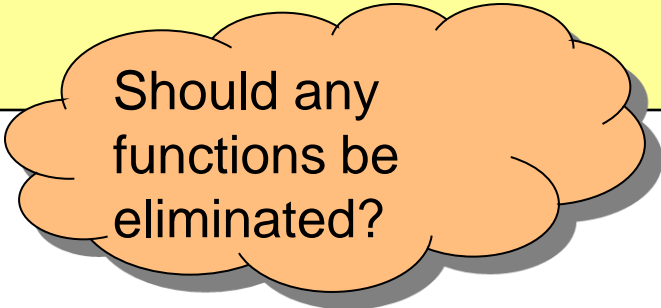
Minimization

- (5) A well-designed module has a minimal interface
 - Function declaration should be in a module's interface if and only if:
 - The function is **necessary** to make objects complete, or
 - The function is **convenient** for many clients
- Why?
 - More functions => higher learning costs, higher maintenance costs

Minimization Example

- Stack

```
/* stack.h */  
  
typedef struct Stack *Stack_T ;  
  
Stack_T Stack_new(void) ;  
void Stack_free(Stack_T s) ;  
void Stack_push(Stack_T s, const char *item) ;  
char *Stack_top(Stack_T s) ;  
void Stack_pop(Stack_T s) ;  
int Stack_isEmpty(Stack_T s) ;
```



Should any
functions be
eliminated?

Minimization Example

- Another Stack function?

```
void Stack_clear(Stack_T s);
```

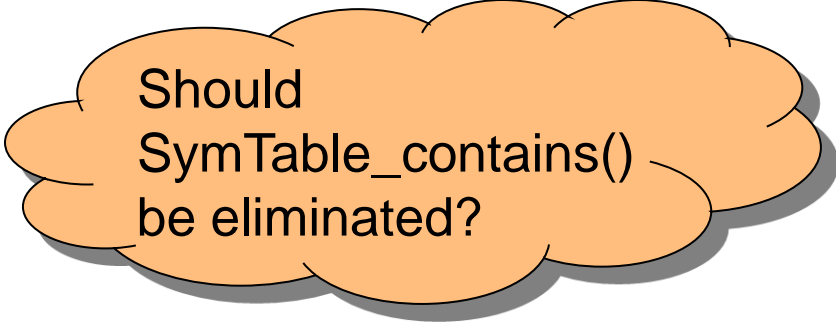
- Pops all items from the Stack object



Should the Stack ADT
define Stack_clear()?

SymTable Aside

- Consider SymTable (from Assignment 3)
 - Declares **SymTable_get()** in interface
 - Declares **SymTable_contains()** in interface



Should
SymTable_contains()
be eliminated?

Error Detection/Handling/Reporting

(6) A well-designed module detects and handles/reports errors

– A module should:

- **Detect** errors
- **Handle** errors if it can; otherwise...
- **Report** errors to its clients

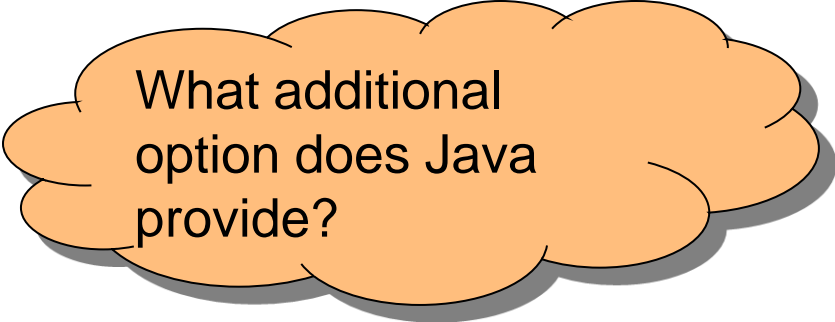
– A module often cannot assume what error-handling action its clients prefer

Detecting and Handling Errors in C

- C options for **detecting** errors
 - **if** statement
 - **assert** macro
- C options for **handling** errors
 - Print message to stderr
 - Impossible in many embedded applications
 - Recover and proceed
 - Sometimes impossible
 - Abort process
 - Often undesirable

Reporting Errors in C

- C options for **reporting** errors to client
 - Set **global variable**?
 - Easy for client to forget to check
 - Bad for multi-threaded programming
 - Use **function return value**?
 - Awkward if return value has some other natural purpose
 - Use extra **call-by-reference parameter**?
 - Awkward for client; must pass additional parameter
 - Call **assert macro**?
 - Terminates the entire program!
- No option is ideal



What additional option does Java provide?

User Errors

Our recommendation: Distinguish between...

(1) **User** errors

- Errors made by human user
- Errors that “could happen”

- Example: Bad data in stdin
- Example: Bad value of command-line argument

- Use **if** statement to detect
- Handle immediately if possible, or...
- Report to client via return value or call-by-reference parameter

Programmer Errors

(2) **Programmer** errors

- Errors made by a programmer
 - Errors that “should never happen”
 - Example: `int` parameter should not be negative, but is
 - Example: pointer parameter should not be **NULL**, but is
 - Use **assert** to detect and handle
- The distinction sometimes is unclear
 - Example: Write to file fails because disk is full

Error Handling Example

- Stack

```
/* stack.c */
...
int Stack_push(Stack_T s, const char *item) {
    struct Node *p;
    assert(s != NULL);
    p = (struct Node*)malloc(sizeof(struct Node));
    if (p == NULL) return 0;
    p->item = item;
    p->next = s->first;
    s->first = p;
    return 1;
}
```

- Invalid parameter is **programmer** error
 - Should never happen
 - Detect and handle via **assert**
- Memory allocation failure is **user** error
 - Could happen (huge data set and/or small computer)
 - Detect via **if**; report to client via return value

Establishing Contracts

- (7) A well-designed module establishes contracts
 - A module should establish contracts with its clients
 - Contracts should describe what each function does, esp:
 - Meanings of parameters
 - Work performed
 - Meaning of return value
 - Side effects
- Why?
 - Facilitates cooperation between multiple programmers
 - Assigns blame to contract violators!!!
 - If your functions have precise contracts and implement them correctly, then the bug must be in someone else's code!!!

Establishing Contracts in C

- Our recommendation...
- In C, establish contracts via comments in module interface

Establishing Contracts Example

- Stack

```
/* stack.h */  
...  
int Stack_push(Stack_T s, const char *item);  
/* Push item onto s. Return 1 (TRUE)  
   if successful, or 0 (FALSE) if  
   insufficient memory is available. */  
...
```

- Comment defines contract:

- Meaning of function's parameters
 - s is the stack to be affected; item is the item to be pushed
- Work performed
 - Push item onto s
- Meaning of return value
 - Indicates success/failure
- Side effects
 - (None, by default)

Summary

- A well-designed module:
 - (1) Separates interface and implementation
 - (2) Encapsulates data
 - (3) Manages resources consistently
 - (4) Is consistent
 - (5) Has a minimal interface
 - (6) Detects and handles/reports errors
 - (7) Establishes contracts