Scope and Blocks

Goals of this Lecture

- Help you learn:
  - Leftover from the last lecture
  - Local vs. global variables, scope, and blocks
- Why?
  - Knowing lifetime and visibility of identifiers is crucial in writing correct code

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 functi ons must share a variable or when a few functions share a large number of variables.
- In most cases, it's better for functions to communi cate through parameters rather than by sharing va riables:
  - If we change an external variable during program mainten ance (by altering its type, say), we'll need to check every f unction in the same file to see how the change affects it.
  - If an external variable is assigned an incorrect value, it m ay be difficult to identify the guilty function.
  - Functions that rely on external variables are hard to reus e 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

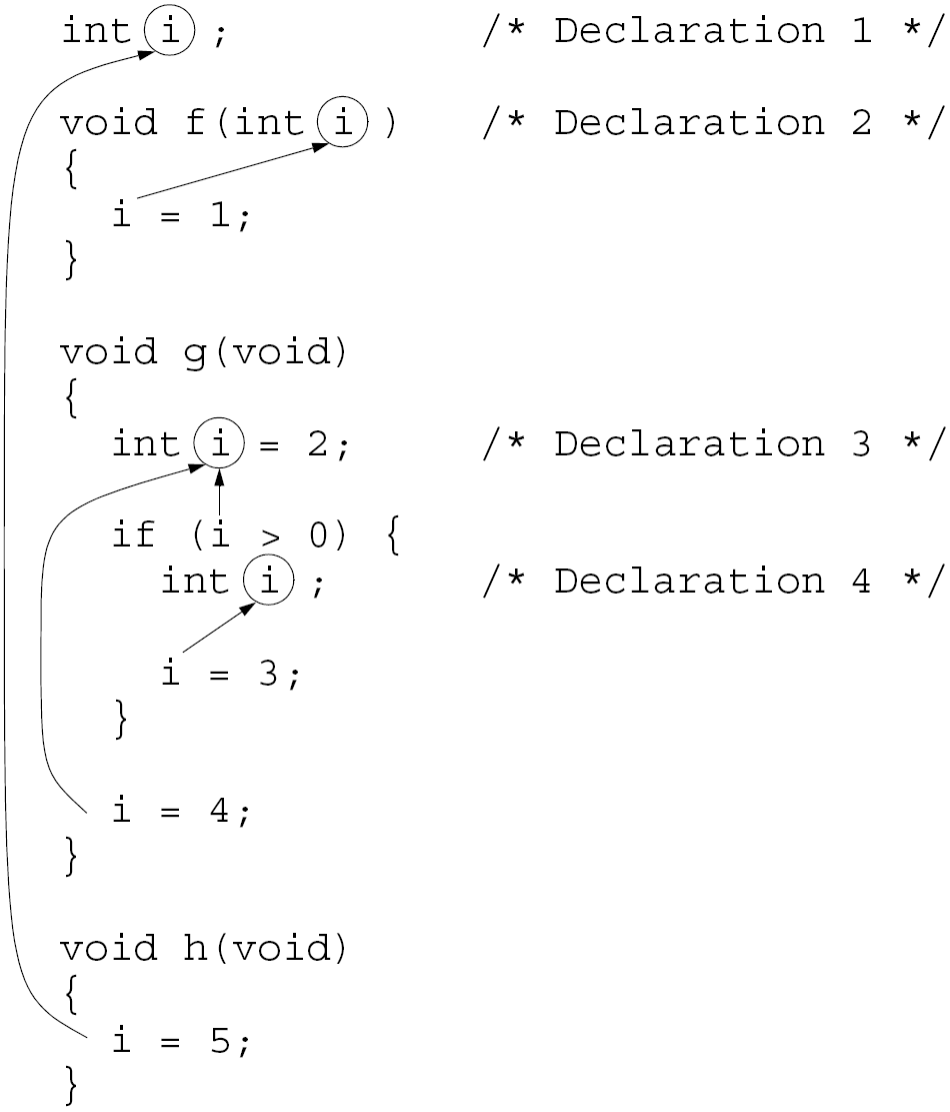
- By default, the storage duration of a variable declared in a block is **automatic**: storage for the variable is allocated when the block is entered and deallocated when the block is exited.
- The variable has block scope; it can't be referenced outside the block.
- A variable that belongs to a block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