Scope of Names

- The scope of a variable determines the region over which you can access the variable by its name.
- C provides four types of scope:
  - Program scope
  - File scope
  - Function scope
  - Block scope

Program Scope

- The variable exists for the program's lifetime and can be accessed from any file comprising the program.
  - To define a global variable, omit the extern keyword, and include an initializer (needed if you want a value other than 0).
  - To link to a global variable, include the extern keyword but omit an initializer
- Example:
  - Variable with program scope is declared and referenced file 1.
  - Variable with program scope is referenced in file 2.

```
/*
 * file 1
 */

int ticks = 1

void tick_tock() {
    ticks += 1;
}
```

```
/*
 * file 2
 */

extern int ticks;

int read_clock() {
    return ticks * TICKS_PER_SECOND;
}
```

File Scope

- The variable is visible from its point of declaration to the end of the source file.
- To give a variable file scope, define it outside a function with the **static** keyword

```
/*
 * file 3
 */

static long long int boot_time = 0;

void at_boot(void) {
    boot_time = get_clock();
}

...

int main(void) {
    printf("%i\n", boot_time);
```

```
/*
 * file 4
 */

/* THE LINE BELOW WILL FAIL
 * when the executable is constructed.
*/

extern long long int boot_time = 0;
```

Function Scope

- The name is visible from the beginning to the end of a function.
- According to the ANSI standard, the scope of function arguments is the same as the scope of variables defined at the outmost scope of a function. Shadowing of function arguments is not allowed.
- (Shadowing of global variables is permitted, however.)

```
/*
 * file 5
 */

void function_f(int x)
{
    ... = x + ...;
}
```

```
/*
 * file 6
 */

/* The variable declaration within the
 * function below will cause a compiler
 * error.
 */
void function_g (int x)
{
   int x;   /* Not possible. */
}
```

```
/*
 * file 7
 */
int sum = 0;

void function_h(int x)
{
    int sum = init_sum();   /* different! */
}
```

Block Scope

- The variable is visible from its point of declaration to the end of the block. A block is any series of statements enclosed by braces.

```
/*
 * file 7
 */

int sum;

void function_y (int X[], int n) {
    int j;

    {
        /* Start of a nested scope */

        int j;
        for (j = 0, sum = 0; j < n; j += 1) {
            sum += X[j];
        }

        /* End of a nested scope */
    }
}
```

File input and output

- C, like most languages, provides facilities for reading and writing files
- files are accessed as **streams** using `FILE` objects
- the `fopen()` function is used to open a file; it returns a pointer to info about the file being opened

```
FILE *data = fopen("input.txt", "r");
```

- streams `FILE *stdin`, `FILE *stdout`, and `FILE *stderr` are automatically opened by the O/S when a program starts

File input and output (2)

- open modes (text): "r" for reading, "w" for writing, and "a" for appending
- open modes (binary): "rb" for reading, "wb" for writing, and "ab" for appending
- the **`fclose()`** function is used to close a file and flush any associated buffers
- use **`fgetc()`** to read a single character from an open file (file was opened in "r" mode)
- similarly, **`fputc()`** will output a single character to the open file (file was opened in "w" mode)

File input and output (3)

```
/* Prints the contents of "data.txt" file, char by char. */

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int ch;
    FILE *data = fopen("data.txt", "r");

    if (data == NULL) {
        fprintf(stderr, "unable to open data.txt\n");
        exit(1);
    }

    while ((ch = fgetc(data)) != EOF) {
        printf("%c", ch);
    }
    fclose(data);

    return 0;
}
```

File input and output (4)

- function **`fread()`** reads n elements of a fixed size from an open stream
  - **`extern size_t fread( void *buf, size_t size, size_t n, FILE *stream );`**
  - returns the number of elements read
- function **`fwrite()`** writes n elements of a fixed size to an open stream
  - **`extern size_t fwrite( void *buf, size_t size, size_t n, FILE *stream );`**
  - returns the number of elements written

File input and output (5)

```
#include <stdio.h>
#define BUFLEN 1024

void process_buffer(char b[], int) {
    /* some code here ... */
}


int main(void) {
    char buffer[BUFLEN];

    FILE *data = fopen("data.txt", "r");

    while (feof(data) != 0) {
        n = fread(buffer, sizeof(char), BUFLEN, data);
        process_buffer(buffer,n);
    }
    fclose(data);

    return 0;
}
```

Operators and Expressions

- arithmetic operators: `+`, `-`
- multiplicative operators: `*`, `/`, `%`
- relational operators: `<`, `<=`, `>`, `>=`
- equality operators: `==`, `!=`
- logical operators: `&&`, `||`, `!`
- bitwise operators: `~`, `<<`, `>>`, `&`, `|`, `^`

Operators and Expressions

- assignment operators: `=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`
  - `x op= expr` is the same as `x = x op expr`
- increment and decrement: `++, --`
- ternary (conditional) operator: `? :`
  - `x = bexpr ? expr_if_true : expr_if_false;`

Operators and Expressions

- comma operator: `x, y`
  - evaluate **`x`**, evaluate **`y`**, result is **`y`**
- cast operator: `(type) expr`
- sizeof operator:
  ```
  sizeof(type)
  sizeof(var)
  ```
- memory operators:
  ```
  &x, *x, x->y,
  x.y, x[5]
  ```

Operator precedence

- Expressions often use several operators
- Order in which operations performed is partially determined by operator precedence
- Also determined by associativity
- Example: "*" and "/" take precedence over "+" and "-"
- Example: "=" has lower precedence than "+", which has lower precedence "*" which has lower precedence than "*" as dereference

```
float disc;

/* ...  */
disc = b * b - 4 * a * c;

/* (b * b) - ((4 * a) * c) */
```

```
float *pf;

/* ...  */

x = y = z = temp + *pf * k;

/* (x = (y = (z =  (temp + ((*pf) * k))))) */
```

Operator precedence

- All C reference manuals will have a table of precedence
  - (or search on Google for "c operator precedence")
- Rule of thumb: From highest to lowest
  1. Primary Expression operators (e.g., "( )", "[ ]", "->", etc.)
  2. Unary operators (*, -, &, ++, etc.)
  3. Binary operators (+, -, &, &&, etc.)
  4. Ternary operator (?:)
  5. Assignment operators (=, +=, etc.)
  6. Comma
- If in doubt: **use parentheses**

Some other operators (not in Java)

- comma operator
  - `x = (e1,e2,…,en)` has the effect of `x = en`
  - `for(i=0, j=0, k=10; bexpr; i+= 1, j+=1) {S}`
- sizeof operator
  - `sizeof(type) or sizeof(variable)`
  - compile-time operator
- memory operators
  - Array element: `x[5]`
  - Member of operator (structs): `x.y, x->y`
  - "contents of" : `*x`
  - "address of": `&x`

C Preprocessor

- The C preprocessor is a separate program that runs before the compiler. The preprocessor provides the following capabilities:
  - macro processing
  - inclusion of additional C source files
  - conditional compilation

Macro processing

- A macro is a name that has an associated text string
  - not type checked by compiler
- Macros are introduced to a program using the **`#define`** directive

```
#define BUFSIZE 512
#define min(x,y) ((x) < (y) ? (x) : (y))
char buffer[BUFSIZE];
int x,y;
…
int z = min(x,y);
```

#include Directive

- You include the contents of a standard header or a user-defined source file in the current source file by writing an include directive:

```
#include <stdio.h>
#include <sys/file.h>
#include "bitstring.h"
```

- (Advice) The quoted form is used for your own '.h' files; the angle bracket form for system '.h' files.

Some Standard Headers

Header file	Contains function prototypes for …
<stdio.h>	The standard I/O library functions and constants/types used by them.
<math.h>	Double-precision math functions and constants (pi, e, ..).
<stdlib.h>	Memory allocation functions and general utility functions.
<string.h>	Functions to manipulate C strings.
<ctype.h>	Character testing and mapping functions.

Conditional Compilation

The preprocessor provides a mechanism to include/exclude selected source lines from compilation:

`#if expr`
`  S1;`
`#elif expr`
`  S2;`
`#else`
`  S3;`
`#endif`
	`#ifdef expr`
	`  S1;`
	`#elif expr`
	`  S2;`
	`#else`
	`  S3;`
	`#endif`
		`#ifndef expr`
		`  S1;`
		`#elif expr`
		`  S2;`
		`#else`
		`  S3;`
		`#endif`
			`#if defined(expr)`
			`  S1;`
			`#elif expr`
			`  S2;`
			`#else`
			`  S3;`
			`#endif`

Conditional Compilation (2)

`#define DEBUG 2`	`#define DEBUG`	`#undef DEBUG`
`#if 1`	`#ifdef DEBUG`	`#ifndef DEBUG`
`// Compile S1`	` S;`	` S;`
` S1;`	`#endif`	`#endif`
`#else`		
`// Not compiled`	`#if defined(DEBUG)`	`#if !defined(DEBUG)`
` S2;`	`// Compile S1`	`// Compile S1`
`#endif`	` S1;`	` S1;`
	`#else`	`#else`
`#if DEBUG == 1`	`// Not compiled`	`// Not compiled`
` S;`	` S2;`	` S2;`
`#endif`	`#endif`	`#endif`

Function Pointers

- In your travels you will see code that looks a bit like the following:
  - "foo = (*fp)(x, y)
  - The function call is actually performed to whatever function is stored at the address in variable "fp"
- Strictly speaking:
  - A function is not a variable…
  - … yet we can assign the address of functions into pointers, pass them to functions, return them from functions, etc.
- A function name used as a reference without an argument is just the function's address

Function pointers

- The variable is visible from its point of declaration to the end of the source file.
- To give a variable file scope, define it outside a function with the **static** keyword

```
/*
 * file 3
 */

static long long int boot_time = 0;

void at_boot(void) {
    boot_time = get_clock();
}

...

int main(void) {
    printf("%i\n", boot_time);
```

```
/*
 * file 4
 */

/* THE LINE BELOW WILL FAIL
 * when the executable is constructed.
*/

extern long long boot_time = 0;
```

Abstract Data Types

- So far, we have described basic data types, all the standard C statements, operators and expressions, functions, and function prototypes.
- We want to introduce the concept of modularization
- Before there were object oriented languages like Java and C++, users of imperative languages used **abstract data types (ADT):**
  – an abstract data type is a set of operations which access a collection of stored data
  – in Java and C++ this idea is called **encapsulation**
- Since ANSI compilers support separate compilation of source modules, we can use abstract data types and function prototypes to ***simulate modules:***
  – this is simply for convenience
  – a C compiler does not force us to use separate files
  – allows us to implement the “one declaration – one definition” rule

Abstract Data Types (2)

- For module **`"mod"`** there are two files:
  - Interface module: named **`"mod.h"`** contains function prototypes, public type definitions, constants, and when necessary declarations for global variables. Interface modules are also called header files.
    - Interface modules are accessed using the **`#include`** C preprocessor directive
  - Implementation module: named **`"mod.c"`** contains the implementation of functions declared in the interface module.

Example: module bitstring

- example: module **`bitstring`**
  - Interface module: **`bitstring.h`** contains the declarations for data structures and operations required to support bitstring manipulation. Contains things which **must** be visible.
    - programmer's responsibility
  - Implementation module: **`bitstring.c`** contains implementation of bitstring operations

Interface Module

```
#ifndef BITSTRING_H
#define BITSTRING_H

typedef unsigned int Uint;
typedef enum _bool { false = 0, true = 1 } bool;

#define BITSPERBYTE      8
#define ALLOCSIZE        (sizeof(Uint)* BITSPERBYTE)
#define BYTESPERAUNIT    (sizeof(Uint))

/* -- Bit Operations */

extern void ClearBits( Uint[], Uint );
extern void SetBit( Uint[], Uint );
extern void ResetBit( Uint[], Uint );
extern bool TestBit( Uint[], Uint );

#endif
```

Implementation Module

```
#include "bitstring.h"

/* Clear a bit string */
void
ClearBits( Uint bstr[], Uint naunits ) {
    Uint i;

    for ( i = 0; i < naunits; i++ )
        bstr[i] = 0;
}

/* Set a bit in a bit string */
void
SetBit( Uint bstr[], Uint bit ) {
    Uint b_index = ( bit - 1 ) / ALLOCSIZE;
    Uint b_offset = ( bit - 1 ) % ALLOCSIZE;

    bstr[b_index] |= ( 1 << b_offset );
}
```

Implementation Module (2)

```
/* Reset a bit in a bit string */
void
ResetBit( Uint bstr[], Uint bit ) {
    Uint b_index = ( bit - 1 ) / ALLOCSIZE;
    Uint b_offset = ( bit - 1 ) % ALLOCSIZE;

    bstr[b_index] &= (~( 1 << b_offset));
}

/* Determine the state of a bit in a bitstring */
bool
TestBit( Uint bstr[], Uint bit ) {
    Uint b_index = ( bit - 1 ) / ALLOCSIZE;
    Uint b_offset = ( bit - 1 ) % ALLOCSIZE;

    return( (bstr[b_index] & (1 << b_offset)) ? true : false );
}
```

Using the Bitstring Module

```c
#include "bitstring.h"

#define NUNITS 4

int main( int argc, char *argv[] ) {
    Uint set[NUNITS];

    ClearBits(set,NUNITS);
    SetBit(set,8);
    SetBit(set,12);

    if (TestBit(set,12) == true)
        ResetBit(set,12);

    return 0;
 }
```