Advanced C Programming

some from Expert C Programming: Deep C Secrets
by Peter van der Linden

CIS*2750

Advanced Programming Concepts

Topics

- Scope of symbol names
  - 4 flavours
- Precedence of operators
  - associativity
  - syntax of declarations with multiple operators

Scope

- **Definition**
  - Region over which you can access a variable by name.
- There are 4 types of scope:
  - Program scope … widest
  - File scope
  - Function scope
  - Block scope … narrowest

Scoping Principle

- Always define a symbol in the *narrowest* scope that works

- Reasons?

1. Program Scope

- The variable is accessible by all source files that make up the executable.
  - In C, all functions
  - Global (**extern**) variables

- How does it work?
  - Initially, talking about language-independent concepts →

Program Symbol Concepts

- Names used for *data* and *functions*
  - variable name, typedef, enum, struct (fields), class (data members, methods), and more
- **Definition:** where the named thing "lives"
  - actual *memory location* of data or function
- **Reference:** some *use* of the thing by name
  - load/store, call: must be "resolved" to location
- **Declaration:** tells compiler *about* the name
  - compiler can verify that references are correct

Examples

int max(int a, int b); // prototype *declaration*

float sum = 0.0; // variable *definition*

sum = **sum***10 + **max**(x,y); // *references*

^*store* ^*load* ^*call*

// function *definition*

int max(int a, int b) { return a>b? a : b; }

// if this was up top before the reference, the definition
// would serve as a declaration, too

External Symbols

- Program scope symbols are passed to **linker** (“man ld”, gcc is front end) in .o file
  - External definition, “extdef”
  - External reference, “extref”
- In linked executable, each external symbol:
  - Exactly 1 extdef, or else…
    - “undefined external” “multiply defined external”
  - Any number of extrefs
    - substituted with final memory address of symbol

“Externals”

- Having “program scope” (external symbols) is a common requirement
  - assembly language
  - all kinds of programming languages
  - allows big program to be linked together out of small modules
- Each language has own convention for designating extdef & extref

Using Program Scope in C

Function

- extdef: `CalStatus` **`readCalFile`**`(FILE *const…){…}`
  - **definition** only appears in **one** .c (calutil.c)
- declaration: `CalStatus` **`readCalFile`**`(FILE *const…);`
  - prototype **declaration** (.h) included in **many** .c files
- extref: `stat =` **`readCalFile`**`(ics);  //call`

- Variable (*don't have any in A1)*
  - extdef: `FILE *`**`inputfile;`**
    - definition only appears in one .c, outside any function
    - can initialize: *type varname* = *initial_value*;
  - declararation: **`extern`** `FILE *`**`inputfile;`**
    - declaration appears anywhere, in/outside functions
  - extref: `fclose(`**`inputfile`**`);`

2. File Scope

- A variable is accessible from its declaration (definition) point to the end of the file.
  - In C, **static** variables.
    - CAUTION: “static” keyword has multiple uses!
- If variable defined outside any function…
  - would normally be “program scope” (global)
  - “static” keyword keeps definition from being passed to linker → doesn’t become *external*

Scope vs. Storage Class

- **Storage class** applies to where & how long variable is kept, not who can access it (=scope)
- So-called “static” storage
  - Exactly one instance of variable in executable program
  - Applies to program scope (global variables), “static” file scope variables, and “static” local variables
- Issue confused in C/C++ (real “deep secret” ☺)
  - Program scope (globals) are static in nature, but without “static” keyword
  - If you add “static” keyword, not global anymore!

Contrast Automatic Storage

- Associated with functions
  - Arguments
  - Local variables inside function
- Fresh temporary copy created **on the stack** every time function called
  - Copy can be initialized (same value each time)
  - Copy goes away when function returns to caller
  - Allows recursion to work!
- "static" keyword changes local variable from automatic to static storage class
  - Initialization effective once, when program started

Dynamic Storage

- Third class of storage, contrasted with static and automatic
- Created (temporarily) **on the heap** via malloc(), calloc(), realloc()
  - Must be explicitly freed via free()
- Address (pointer) has to go in some variable
  - *That* variable has scope and storage class itself

3. Function Scope

- Accessible *throughout* a function.
  - In C, only **goto** labels have function scope; therefore *you* will never see them ☺
  - "Throughout" means you can jump ahead:

```
goto bummer;
…
bummer: printf( "Outta here!");
```

4. Block Scope

- The variable is accessible *after* its declaration point to the end of the block in which it was declared.
  - In C, these are **local** variables.
  - Includes function's parameters

Example (all in one .c file)

```
int i;                  /* Program scope */
static int j;           /* File scope */
func( int k ) {         /* func is program scope,
                              k is block scope */
  int m;                /* Block scope */
  ….
}
```

/* which are static **storage**, auto? */

What Happens?

```
func() {
    int a = 11;
    {
        int b = 10;
    }
    printf ( “%d %d\n”,a,b);
 }
```

Won’t work!

The variable *b* is inside a block and therefore is not **visible** to the rest of the function.

What Happens?

```
newfunc() {
    int a = 11;
    {
        int b = 10;
        printf ( “%d\n”,b);
    }
    printf ( “%d\n”,a);
 }
```

WORKS!

Precedence of Operators

- **Definition**
  - Determines the *order* in which operators are evaluated.

$$x = 25 * a + c / 2.1$$

  - Operators are used to calculate values for both numeric and pointer expressions
- Operators also have an **associativity** which is used to determine which operands are grouped with similar operators.

Associativity

- Applies with 2 or more operators of **same precedence**:

$$A\ op_1\ B\ op_2\ C\ op_3\ D$$

  - Answers question: Which *op* is done first?
- Associativity can be either Left-to-Right or Right-to-Left

Associativity

- **Left-to-Right** (=left associative) is most common

  a + b – c;

  – The + and – operators are both evaluated left-to-right so the expression is

    "a plus b, then subtract c"

  – Equivalent to: (a + b) – c;

Associativity

- **Right-to-Left** (=right associative) is rare

  a = b = c;

  – This expression is read

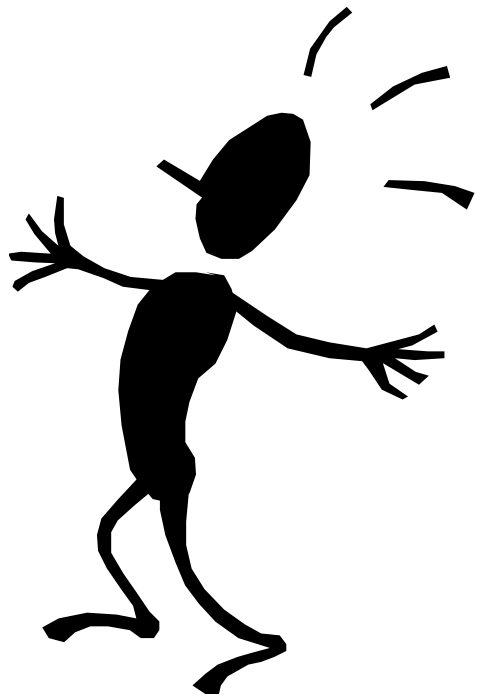
  “assign c to b, then to a”

  – Equivalent to: a = (b = c);

  – Only meaningful because in C, assignment operator is an *expression*, resulting in a *value*

Problems with Precedence

- The precedence of some operators produces problems when they create behaviours which are unexpected.



Problems with Precedence

- Pointer to structure: *p.f
  - **Expectation**: the member f of what p points to

    (*p).f
  - **Actually**: p.f gives a compile error, means:

    *(p.f)
  - **Why?** **.** is higher precedence than *
  - **Note**: The -> operator was made to correct this.

    p->f

Problems with Precedence

int *ap[]

- **Expectation**: ap is a ptr to an array of ints

  int (*ap)[]

- **Actually**: ap is an array of pointers-to-int

  int *(ap[])

- **Why?** [] is higher precedence than *
- **Note**: usually found in declarations.

Problems with Precedence

int *fp()

- **Expectation**: fp is a ptr to a function returning an int

  int (*fp)()

- **Actually**: fp is a function returning a ptr-to-int

  int *(fp())

- **Why?** () is higher than *
- **Note**: usually found in declarations.

Problems with Precedence

c = getchar() != EOF

- **Expectation**: (c = getchar()) != EOF
- **Actually**: c = (getchar() != EOF)
  - c is set equal to the true/false value
- **Why?** comparators == and != have higher precedence than assignment

Solution to Precedence Problems

- When in doubt…

Use parentheses!!!

- Better still, use parentheses anyway
  - *You* may not be in doubt, but the next reader could be