

Information Processing 2 08

scope rules, block structure
static variables
header files
the C preprocessor

Ian Piumarta

Faculty of Engineering, KUAS

last week: functions and program structure

function to search file for string

flowchart, pseudocode, code

factoring the program

function `getchars()`

function `strindex(s, t)`

returning non-integers

`atof()`

using header files to ensure function type safety

simple calculator using `atof()`

desk calculator: stack

Chapter 4. Functions and Program Structure

4.1 Basics of Functions

4.2 Functions Returning Non-integers

4.3 External Variables

this week: program structure, multiple source files

declarations in header files

compilation from multiple files

`static` variables

block structure

defining variables within blocks

variable initialisation

recursive functions

`qsort()`, `swap()`

`#include`, `#define`, `#undef`, conditional compilation

4.4 Scope Rules

4.5 Header Files

4.6 Static Variables

4.7 Register Variables

4.8 Block Structure

4.9 Initialization

4.10 Recursion

4.11 The C Preprocessor

declarations and definitions of functions and global variables

a *declaration* tells the C compiler that something exists: how to *use* it

- a function declaration allows it to be called
- a variable declaration allows it to be accessed (read/written)

a *definition* says what a variable contains or how a function is implemented

- a function definition provides a block of statements that implement the function
- a variable definition can provide initial contents of the variable
- a variable definition also sets aside *storage* (memory) to hold the value

for any given name, all declarations and definition must be equivalent

- for a function name, the prototypes must all be the same
- for a variable name, the type (and size if it is an array) must be the same

a variable or function can be declared any number of times, but **defined only once**

a variable that is declared but never defined will be defined implicitly

scope and local/global variable names

the *scope* of a variable name is *the part of the program in which it can be used*

local variables (and function parameters)

- their scope is from their declaration to the **end of their block**
- (also called *automatic* variables because they vanish at the end of their block)

global variables

- their scope is from their declaration to the **end of the file**
- (also called *external* variables because they are not inside any function)

a global variable can be declared in more than one file

- the **same name** will refer to the **same variable** in all files
- you cannot have two different global variables with the same name

example of global scopes

in this program...

```
int main() { ... }           // stack, sp, push, pop are all out of scope
double stack[STACKMAX];      // variable declaration: scope is from here to end of file
int    sp = 0;               // variable definition: scope is from here to end of file
void push(double n) { ... }  // function definition: stack, sp, and push are in scope
double pop(void) { ... }     // function definition: stack, sp, push, pop are in scope
```

`stack` and `sp` can be used in `push` and `pop`

- they are declared or defined *before* `push` and `pop` are defined
- they are *in-scope* when `push` and `pop` are defined

`push` and `pop` cannot be used in `main`

- they are neither declared nor defined before `main`
- they are *out-of-scope* when `main` is defined

example of function declarations

adding *forward declarations* for `push` and `pop` makes them in-scope in `main`

```
void push(double); // function declaration: push is in-scope in the rest of the file
double pop(void);  // function declaration: pop is in-scope in the rest of the file

int main() {        // push() and pop() can be used (but stack and sp cannot be used)
    ...
    case '+': push(pop() + pop()); break; // push() and pop() are in-scope
    ...
}

double stack[STACKMAX];
int     sp = 0;

void push(double n) { ... } // definition prototype must match earlier declaration
double pop(void) { ... }   // definition prototype must match earlier declaration
```

local variables: function parameters

parameters are local variables that exist only while a function is executing

- they are created when the function begins executing
- when the function returns, they are automatically destroyed
- they are therefore 'automatic' variables whose scope is the entire function body

```
int abs(int n) // variable n exists only while the function is executing
{
    if (n < 0) return -n; // space for n is reclaimed during return
    return n;           // space for n is reclaimed during return
}
```

local (automatic) variables in blocks

any block can contain variable declarations and definitions

- the variables are in-scope within the block (including nested blocks within it)
- they are automatic and are destroyed at the end of the block

```
void table(int width, int height) // width and height in-scope for whole function
{
    int y;                        // y in-scope until end of this block
    for (y = 1; y <= height; ++y) {
        int x;                   // x 'created' here, in-scope to end of this block
        for (x = 0; x < width; ++x)
            print("%4d", x * y);
        print("\n");
    }                             ← // x is out-of-scope after this '}'
    print("%d\n", y); // y in-scope, but cannot print x here (out of scope)
}                               ← // y, height, and width are out-of-scope after this '}'
```

local (automatic) variables

```
int main()
{
    int    i = 42,  j = 666;
    double d = 4.2, e = 6.66;

    printf("%d %d\n", i, j);
    {
        int _temp = i;      // _temp is local to this block only
        i = j;
        j = _temp;
    }
    printf("%d %d\n", i, j);
    printf("%f %f\n", d, e);
    {
        double _temp = d;  // _temp is local to this block only
        d = e;
        e = _temp;
    }
    printf("%f %f\n", d, e);
    return 0;
}
```

the vertical bars on the left show the scopes of the `_temp` variables

nested scopes

a block can be used anywhere that a statement is allowed

- in loops and conditional statements
- anywhere inside another block instead of some other kind of statement

an inner block appearing inside another is said to be 'nested' within the outer block

variables that are in-scope in an outer blocks are also in-scope in the inner block

```
{                                // start of outer block
  int x = 6;                     // scope of x begins
  int y = x + 1;                 // scope of y begins
  {                              // start of nested inner block
    int z = x * y;               // x, y, z are all in-scope now
    printf("%d\n", z);
  }                             ← // scope of z ends here
  printf("%d\n", x);             // x, y are still in scope
}                               ← // scopes of x and y end here
```

name shadowing

the names of local variables must be unique within a block

- you cannot re-declare a name more than once

names from an outer block can be re-declared within an inner block

- the declaration in the inner block *shadows* the outer declaration
- the outer block variable is out-of-scope while the inner block variable is in-scope

```
{                                // start of outer block
    int x = 6;
    int y = x * 7;
    printf("%d\n", y);           // → 42  outer y is in-scope
    {                            // start of nested inner block
        int y = x + 7;           // new declaration of y shadows outer declaration
        printf("%d\n", y);       // → 13  inner y is in-scope
    }                            ← // inner y goes out-of-scope here
    printf("%d\n", y);           // → 42  outer y is back in-scope
}
```

initialisation of global and local variables

global variables can be initialised when they are declared

the initial value is calculated at **compile time**

the initial value *must* be a **constant expression**

```
int sp = 0; // declare and define initial value of sp
```

local variables are initialised every time control passes them during execution

their initialiser values are calculated at **run time**

the initial value can be **any expression**

```
void trimBefore(char string[], char prefix[]) {  
    int from = strindex(string, prefix); // non-constant initialiser  
    if (from > 0) {  
        int to = 0;  
        while ((string[to++] = string[from++]))  
            ;  
    }  
}
```

splitting large programs into separate files

the C compiler can create a program by combining several source files

consider `findstr` that prints lines matching a string, split into multiple files, where

- `getchars.c` contains `int getchars(char line[])`
- `strindex.c` contains `int strindex(char s[], char t[])`
- `main.c` contains `main()`
 - a program that prints all input lines containing the word `LINEMAX`

```
$ cc -o findstr main.c getchars.c strindex.c
```

```
$ cat main.c | ./findstr
```

```
#define LINEMAX 1000
char line[LINEMAX];
while (getchars(line, LINEMAX) >= 0)
    if (strindex(line, "LINEMAX") >= 0)
```

splitting large programs into separate files

main.c must have declarations for the functions in getchars.c and strindex.c

- otherwise it would not be able to call these functions safely

```
#include <stdio.h>

int getchars(char line[], int limit);
int strindex(char s[], char t[]);

int main() { ... }
```

getchars.c uses EOF and so must also include stdio.h

```
#include <stdio.h>

int getchars(char line[], int limit) { ... }
```

strindex.c needs neither, because strindex() has no external dependencies

```
int strindex(char s[], char t[]) { ... }
```

putting declarations into header files

it is important to keep function declarations and definitions consistent

- consider changing the order of parameters of `getchars()`
- the declaration in `main.c` should change too
- if not, the program will crash because `getchars()` is called incorrectly

two solutions: one header file for whole program, or one header file per source file

one header file for whole program

- place all declarations in header file, include it in all files of the program
- simple to manage, not easily reusable (header file is specific to the program)

one header file per source file

- include the header file any files that uses its functions or other definitions
- include the header file in its own source file, to ensure consistency
- harder to manage, but easy to reuse (header file describes one source file)

option 1: single header file for the whole program

the file `findstr.h` could contain

```
#define LINEMAX 1000

int getchars(char line[], int limit);
int strindex(char s[], char t[]);
```

every source file then includes that file

```
#include "findstr.h"    // use "." instead of <...> for your own header files
```

this does two things:

- provides external declarations to files that need them
- ensures declarations in `findstr.h` are always consistent with source files

note the use of quotes `"findstr.h"` which looks in the current directory for `findstr.h` (using angle brackets `<stdio.h>` looks in standard system directories for `stdio.h`)

option 2: one header file per source file

each .c file has a corresponding .h file describing its contents

e.g., getchars.h could contain:

```
int getchars(char line[], int limit);
```

strindex.h could contain:

```
int strindex(char s[], char t[]);
```

then these header files are

- included in any source file than has to call `getchars()`
- included in their own .c file to ensure declarations and definitions are consistent

in getchars.c:

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

in strindex.c:

```
#include "strindex.h"
```

and in main.c:

```
#include <stdio.h>  
#include "getchars.h"  
#include "strindex.h"
```

example: splitting 'reverse Polish' calculator into multiple files

the calculator program from last week has several independent parts

it could be broken up as follows:

- `stack.c` implements `push()` and `pop()`
- `getch.c` implements `getch()` and `ungetch()`
- `getop.c` implements `getop()`
- `atof.c` implements `atof()`
- `main.c` implements `main()` containing the actual calculator algorithm

each source file still has to `#include` declarations it depends on

- `stdio.h` for `EOF` and `printf()`
- `ctype.h` for `isdigit()` and `isspace()`

example: splitting 'reverse Polish' calculator into multiple files

`main.c` has to declare all the external functions that it needs

```
#define NUMBER '0'

double atof(char string[]);
int     getop(char number[]);
void    push(double value);
double pop(void);
```

`main.c` and `getop.c` must agree on a special 'operator' character `NUMBER`

- representing a number being transferred to the character array

variables, function, definitions not used outside their file do not need to be declared

- e.g., `STACKMAX`, `stack`, and `sp` in `stack.c` are 'private' to that file

making global variables private using static

`stack` and `sp` in `stack.c` are 'private' to that file

- they are not meant to be accessed by any other file

a lazy (or malicious) programmer might try to access them from code in another file

also, their names are accessible outside their file, so name conflicts are possible

- variables with the same name cannot be used in another file

a `static` variable or function has its scope limited to its source file

- it cannot be accessed from any other file
- not even by declaring it as an external variable

in `stack.c`:

```
static double stack[STACKMAX]; // only accessible in this file
static int    sp = 0;          // only accessible in this file
```

making local variables private using static

`static` can also be used on automatic variables within a function

a `static` local variable becomes external instead of automatic

- it behaves like a global variable
- its initialiser (if present) must be a constant expression
- its value is preserved between function calls

the scope of the variable is still the block in which it is declared

- use this to make a 'global' variable private to a block or function body

in this example, `total` is external but in-scope only within the function body

```
void accumulateAndPrint(int n) {  
    static int total = 0; // initialisation happens once, like a global variable  
    total += n;           // value of total is preserved between function calls  
    printf("accumulated value is now %d\n", total);  
}
```

cpp, the C preprocessor

how do `#include` and `#define` really work?

source files are *preprocessed* before they are compiled

the preprocessor performs textual transformations on the source file

we have already seen two:

- `#include` inserts the contents of a file during compilation
- `#define` replaces a name by an arbitrary sequence of characters

two other useful preprocessor features are:

- macros with arguments (which behave a bit like functions)
- conditional compilation (ignoring sections of the program source)

cpp: file inclusion

often used to include a file of definitions or declarations

a line having one of these forms

```
#include "filename"
```

```
#include <filename>
```

is replaced by the contents of the file called *filename*

the first form (using `"..."`) looks for *filename* in the same directory as the source file

the second form (using `<...>`) looks for *filename* in a system directory

the included file is often a *header* file containing declarations and macro definitions

- but you can include a `.c` file in your program too, if you want to

cpp: macro substitution

macro definitions have the form

```
#define  name  replacement text
```

after this, each time *name* appears it is replaced by *replacement text*

- *name* must follow the rules for variable names
- *replacement text* can be anything you want
- *replacement text* stops at the end of the line
 - a long definition can cover several lines by placing \ at the end of each line

(preprocessor-defined *names* are also called *symbols*)

```
#define  STACKMAX  32
```

```
#define  forever    for (;;) /* infinite loop */
```

cpp: macro substitution

macros can also have named arguments

after defining

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

the line

```
x = MAX(p+q, r+s);
```

will be replaced by

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

macros can be undefined when no longer wanted

```
#undef STACKMAX
```

```
#undef SQUARE
```

cpp: inventing new keywords

the `if` statement checks whether something is *true*

if you need to check whether a lot of things are *false*, you really want `unless`

```
#define unless(X)      if (!(X))

int main()
{
    for (int i = 0; i < 10; ++i)
        if (i % 2 == 0) printf("%d\n", i);    // even

    for (int i = 0; i < 10; ++i)
        unless (i % 2 == 0) printf("%d\n", i);    // odd
}
```

obviously, do this very sparingly or your readers won't understand your programs

cpp: macro pitfalls

use parentheses liberally to protect arguments that might be expressions

```
#define SQUARE(X)  X * X           // wrong!
z = SQUARE(y+1);                  // z = y + 1*y + 1;

#define SQUARE(X)  (X) * (X)       // wrong!
z = 2.0 / SQUARE(y);               // z = 2.0 / (y) * (y);

#define SQUARE(X)  ((X) * (X))     // right (☺)
z = 2.0 / SQUARE(y+1);             // 2.0 / ((y+1) * (y+1))
```

cpp: conditional compilation

`#if` is used to conditionally include parts of the source file

the general form is

```
#if  constant expression
...  // included if constant expression was non-zero
#elif another expression
...  // included if another expression was non-zero
#else
...  // included if all expressions were zero
#endif
```

constant expression cannot include `enum` or other named constants

- but it can include `#defined` names, which will be replaced properly

`#elif` behaves like “else if”, and the `#elif` and `#else` parts are optional

cpp: conditional compilation

within `#if`, the function `defined(name)` is 1 if `name` is defined, otherwise 0

to make sure a header file `mydecls.h` is only included once in a program, you can use

```
#if !defined(_mydecls_h)
#define _mydecls_h
```

*first line of file
sentinel, preventing re-inclusion*

```
/* contents of mydecls.h go here */
```

```
#endif
```

last line of file

note the following shorter forms:

<code>#ifdef X</code>	is equivalent to	<code>#if defined(X)</code>
<code>#ifndef X</code>	is equivalent to	<code>#if !defined(X)</code>

cpp: conditional compilation

the following header file `minmax.h` is safe to include more than once

- the second and subsequent times would not redefine `MIN` and `MAX`

```
#ifndef _minmax_h                true only the first time this file is included
#define _minmax_h

#define MIN(X, Y)  ((X) < (Y) ? (X) : (Y))
#define MAX(X, Y)  ((X) > (Y) ? (X) : (Y))

#endif // _minmax_h
```

the name of the ‘sentinel’ symbol you use is not significant

- try to choose one that is unlikely to be used elsewhere
- I usually use the name of my header file with `_` in front and `.` changed to `_`
e.g., `foobar.h` would test `#ifndef _foobar_h`

next week: memory, addresses, arrays and pointers

memory, addresses, variables, pointers, indirection

`swap(a, b)` that actually works

`getint()`

equivalence of arrays, elements, and pointers

`strlen()` using pointers

passing sub-arrays as arguments

negative indices; address arithmetic

`alloc()` and `afree()`

char array vs. char pointers vs. strings

versions of `strcpy()`, `strcmp()`

Chapter 5. Pointers and Arrays

5.1 Pointers and Addresses

5.2 Pointers and Function Arguments

5.3 Pointers and Arrays

5.4 Address Arithmetic

5.5 Character Pointers and Functions

assignment

please download assignment from
MS Team “Information Processing 2”, “General” channel

exercises

variable scope

single header file for entire program

one header file per source file

including multiple header files

truncating a string after a target substring

preventing multiple inclusion of header file

macro safety

home-made control constructs

swapping variables using macro and block

making the swap macro robust in all contexts

13:00 5 last week: functions, program structure
13:05 5 this week: program structure, multiple source files
13:10 5 global declarations and definitions
13:15 5 examples
13:20 5 function parameters
13:25 5 variables in blocks
13:30 5 nested scopes
13:35 5 name shadowing
13:40 5 using multiple source files
13:45 5 declarations in header files
13:50 5 single header for whole program
13:55 5 one header per source file
14:00 5 splitting calculator into multiple files
14:05 5 initialisation of locals and globals
14:10 5 static variables and functions
14:15 5 cpp: file inclusion
14:20 5 cpp: macro substitution
14:25 5 cpp: macro pitfalls
14:30 10 *break*
14:40 5 cpp: conditional compilation
14:45 5 cpp: protecting against multiple inclusions
14:50 80 exercises
16:10 00 end