

Functions and Program Structure

CSE 2031
Fall 2010

24 October 2010

1

Function Basics (4.1)

- Functions

- Break large computing tasks into smaller: During the design stage try to separate small tasks that may be implemented as single function.
- Can be reused

- Scope

- Where the name can be used/visible?

2

Definition and Declaration

- Declaration

```
returned_type function_name ( list_of_arguments ) ;
```

- Definition

```
returned_type function_name ( list_of_arguments )  
{  
    declarations and statements  
}
```

- Return statement

```
return expression ;
```

3

return Statement (4.2)

- Functions use **return** statement to return the result to the caller.
return *expression*;
- Functions can return arbitrary type: **void**, **int**, **double**, pointer (to the variable or function), etc.
- Inconsistent expressions will be cast to the returned type of the function.

4

External Variables (4.3)

- Internal variables
 - Defined inside of the function body and exists only when the function is executed.
- External objects
 - External variables and function are defined outside of any function.
 - External variables may be used as a tool to communicate between functions.

5

External Variables (cont.)

- Problems with external variables
 - Everyone can access the variable (like *public* variables in Java).
 - Low level of control.
 - Too many externals leads to bad program structure with too many data connections between functions (problem with modularity and reusing).
- Bottom line:
 - **Avoid external variables whenever possible!**

6

Scope (4.4)

- Scope is a part of the program within which a declared name can be used.
- Questions of interest:
 - How to write declarations so that variables are properly declared during compilation?
 - How are declarations arranged so that all the pieces will be properly connected when program is loaded?
 - How are declarations organized so there is only one copy?
 - How external variables are initialized (so that all of them are initialize once)?

7

extern Declaration

In order to use a variable in another file or before its definition.

`file1.c`

```
extern int size ;  
extern char buff[ ] ;
```

`file2.c`

```
int size = SIZE ;  
char buff[SIZE] ;
```

8

Declaration vs. Definition

Declaration: announces the properties of a variable (type).

Definition = Declaration + Storage to be set aside.

file 1.c

```
extern int sp;  
extern double val[ ];
```

file2.c

```
int sp = 0;  
double val[MAXVAL];
```

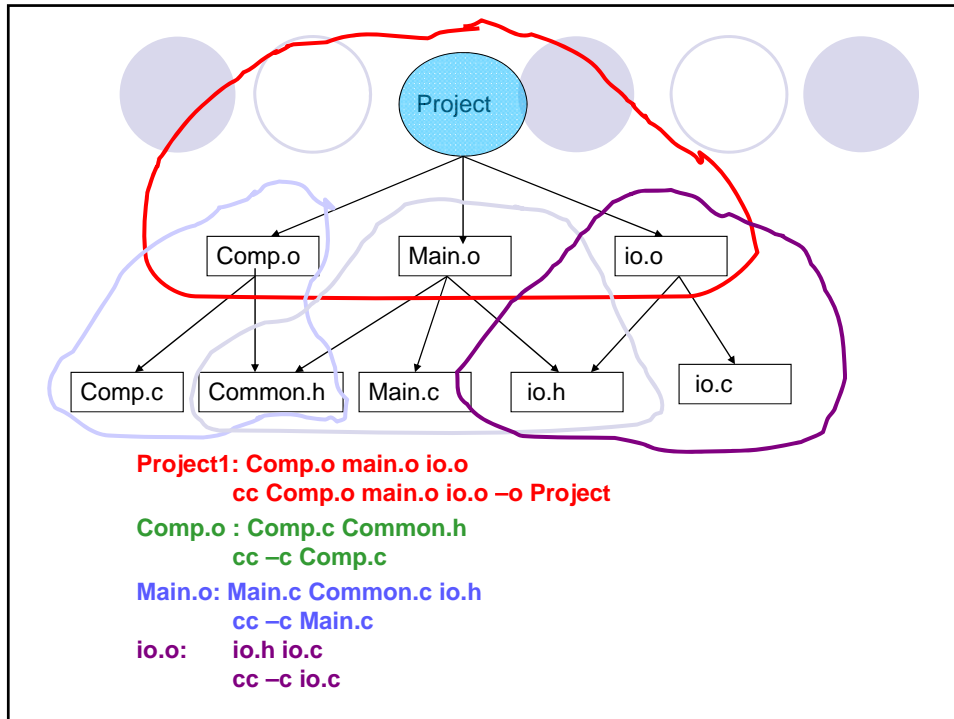
9

Static Variables (4.6)

- **static** declaration restricts (hide) the visibility (scope) of a variable or a function.
- **Static external variables:** visible only in the source file in which they are defined.
- Routines in Comp.c and Main.c cannot access `buf[]` or `bufp`.
- Those variable names can be used in Comp.c and Main.c for other variables without any conflict since they are not visible outside io.c.

```
/* File io.c */  
  
#include <stdio.h>  
  
static char buf[BUFSIZE];  
static int bufp = 0;  
  
int getch(void) { ... }  
  
void ungetch(int c) { ... }  
  
/* end of file io.c */
```

10



Static Variables (cont.)

- **Static function:** its name is invisible outside of the file in which it is declared.
- Note: function names are normally global.
- **Static internal variable:**
 - visible only inside the function in which it is defined.
 - remains in existence (not coming and going each time the function is called).
 - provide private, permanent storage within a single function.

```
static int power
(int base, int n) {
    ...
}

int getline(char s[]) {
    static int counter;
    int next;
    ...
}
```

Register Variables (4.7)

- **register** declaration advises the compiler that the variable will be heavily used.
- The register variable will be placed in machine registers
⇒ smaller, faster program.
- Compilers are free to ignore the advice.
- Examples:

```
register int i;  
register char c;  
my_func( register long x, register unsigned y )
```

13

Register Variables (cont.)

Restrictions due to underlying hardware:

- Only a few registers available.
- Only certain types are allowed.
 - Excess/disallowed declarations are treated as normal variables.
- Not possible to take address of a register var.

14

Block Structure (4.8)

- Block: delimited by { and }

```
if (n > 0) {  
    int i; /* declare a new i */  
    for (i = 0; i < n; i++)  
        ...  
}
```

- `i` is initialized *each time* the block is entered.
- A **static** variable is initialized only the *first time* the block is entered.

15

Block Structure (cont.)

```
int x;  
int y;  
...  
f(double x)  
{  
    double y;  
}
```

- The above works, but avoid that programming style!

16

Initialization (4.9)

Explicit initialization

- External and static vars: constant expressions
- Automatic and register vars: any expressions (involving pre-defined values or function calls)

```
int x = 1;
char squota = '\\';
long day = 1000L * 60L * 60L *
    24L; /* milliseconds/day */

int binsearch(int x, int v[],
    int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

17

Initialization (cont.)

No explicit initialization

- External and static vars: initialized to 0.
- Automatic and register vars: undefined (garbage) initial values.

Arrays:

```
int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
    30, 31 }; /* array of size 12 */
int months[12] = { 100, 25, 75 }; /* the rest is 0*/
char pattern = "ould"; /* same as below */
char pattern[] = { 'o', 'u', 'l', 'd', '\\0' };
```

18

Recursion (4.10)

- In C, a function may call itself either directly or indirectly.

```
#include <stdio.h>

/* print n in decimal */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

19

Recursion (cont.)

- Advantages:
 - More compact code
 - Easier to write and understand
- Disadvantage: more overhead for recursive function calls
 - Stack
 - Parameter saving and returning

20

Macro Substitution (4.11.2)

- `#define name replacement text`
 - subsequent occurrences of the token name will be replaced by the *replacement text*.
- Examples:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
x = max(p+q, r+s);
/* x = ((p+q) > (r+s) ? (p+q) : (r+s)); */
i =1; j = 10;
y = max(i++, j++); /* final values of i and j ?*/

#define square(x) x * x /* what's wrong? */
z = square(y + 1);
```

21

Next time ...

- Big lab test 1 (10% of overall grade)

22