Functions (cont'd): Scope and Lifetime of Variables Recursive Functions

CS2023 Winter 2004

Outcomes: Functions (cont'd)

- "C for Java Programmers", Chapters 7 (section 7.3 – you don't need to read 7.4-7.7); also see other books on C on reserve in the library
- After the conclusion of this section you should be able to
  - distinguish the scope and lifetime of local and global variables in a C program
  - Modify the lifetime of a local variable using `static`
  - Use recursion in C

Scope and Lifetime

- **Scope** of a variable is the region of the program in which it is visible
- The **lifetime** of a variable is the period of time during which memory is allocated to the variable
- Since storage is freed in the reverse order of allocation, a *stack* is a convenient data structure to represent it with
    - (the **run time stack**)
- C's scope rules use *files* (Java uses classes).

Local Variables

```
int log2(int n)
{
  int log = 0; /* local variable */

  while(n > 1) {
    n /= 2;
    log++;
  }
  return log;
}
```

Local Variables

- Variables declared in the body of a function or block

```
int f() {
int i;

…
}
```

- Local variables have
  - automatic storage duration
  - block scope

Global Variables

- **Global variables** are defined outside the body of functions in the file. *Scope* starts at point of definition and *lifetime* same as main.

```
int flag = 0; /* global */
int f() {
…
}
int out = 1; /* global */
int main() {
...
}
```

What Gets Printed?

```
int a=1, b=2, c=3;    /* global variables */
int f();

int main()
{
  printf("%3d\n”, f());
  printf(“%3d%3d%3d\n”, a, b, c);
  return 0;
}

int f(){
  int b, c;             /* b and c are local */
  a = b = c = 4;        /* global b,c are masked */
  return (a + b + c);
}
```

Global Variables

- Global variables should be used with caution, and always carefully *documented*.
- Changing the value of a global variable as a result of calling a function should be avoided; these side-effects make testing, debugging, and in general maintaining the code very difficult.

What Gets Printed?

```
int i;
void print_row();
void print_matrix();
int main()
{
        print_matrix();
        return 0;
}
void print_row()
{
  for (i = 1; i<= 10; i++)
    printf(“*");
}
void print_matrix()
{
  for (i = 1; i<= 10; i++){
    print_row();
    printf(“\n");
  }
}
```

Initialization of Variables

- at compile time:

```
    const int a = 3 * 44;
```

- at run time:

```
    double x = sqrt(2.66);
```

- The value of a *local* variable that is declared, but not initialized, is undefined.

- Global variables are initialized to a "zero" value.

Changing Storage Duration

- **`Static`** storage class for ***local*** variables (declared *inside* a block or function) - the lifetime of the entire program:

```
void login() {
    static int counter = 0;
    counter++;
    ..
}
```

Changes only *lifetime* not *scope*, therefore `counter` not visibile outside function

Organizing a C Program

Preprocessor directives such as `#include` and `#define`

Type definitions (`typedef`)

Declarations of functions and global variables

Function Definitions (beginning with `main`)

- Each doesn't take effect until the line at which it appears

- Recommended order:

  #include directive
  #define directives
  Type definitions
  Declaration of global variables
  Prototypes for functions other than `main`
  Definition of `main`
  Definition of other functions

Recursive Functions

```
int sum(int n)
{
  if (n <= 1)  /* base case */
    return n;
  else
    return (n + sum(n - 1));
}
```

Function call	**Value returned**
sum(1)	1
sum(2)	2 + sum(1) or 2 + 1
sum(3)	3 + sum(2) or 3 + 2 + 1
sum(4)	4 + sum(3) or 4 + 3 + 2 + 1

```
/* write a line backwards */
void write_it();

int main()
{
  printf(“Input a line: “);
  write_it();
  printf(“\n”);
  return 0;
}
void write_it()
{
  int c; /* each call has its own local storage */
  if ((c = getchar()) != '\n')
    write_it();
  putchar(c);
}
```