Lecture07: Strings, Variable Scope, Memory Model
4/8/2013

Slides modified from Yin Lou, Cornell CS2022: Introduction to C

1

Outline

- Review pointers
- New: Strings
- New: Variable Scope (global vs. local variables)
- New: C Memory model

- **Midterm exam next week!**

2

Recap: Pointers

- `int *ptr;`
- Pointers are variables that store memory address of other variables
- Type of variable pointed to depends on type of pointer:
  - `int *ptr` points to an integer value
  - `char *ptr` points to character variable
  - **Can cast between pointer types: `myIntPtr = (int *) malloc(sizeof(..)*..);`**
  - **`void *ptr` has an unspecified type (generic pointer); must be cast to a type before used**

3

Recap: Pointers

- Two main operations
  - * dereference: get the value at the memory location stored in a pointer
  - & address of: get the address of a variable
  - `int *myPtr = &myVar;`
- Pointer arithmetic: directly manipulate a pointer's content to access other locations
  - Use with caution!: can access bad areas of memory and cause a crash
  - However, it is useful in accessing and manipulating data structures
- Can have pointers to pointers
  - `int **my2dArray;`

4

Why Pointers – change values of variable(s) passed into functions

```
void set_to_zero(int a)
{
   a = 0;
}

void main()
{
   int a;
   a = 5;
   set_to_zero(a);
   printf("%d\n", a);
}
```

```
void set_to_zero(int *a)
{
   *a = 0;
}

void main()
{
   int a;
   a = 5;
   set_to_zero(&a);
   printf("%d\n", a);
}
```

5

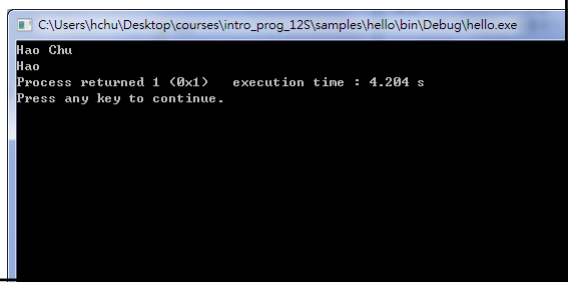
Strings

- There is no string type in C!
- Instead, strings are implemented as arrays of characters:
  - `char*` or `char []`
- Enclosed in double-quotes
  - Terminated by NULL character **(‘\0')**
  - `"Hello"`
- same as
  - `char str[] = {'H', 'e', 'l', 'l', 'o', ‘\0'}`

6

Strings: printf() and scanf() placeholder %s

```
int main()
{
    char name[10];
    scanf("%s", name); // read one word, skip spaces, newlines, tabs, etc.
    printf("%s", name);

    char months[12][10] =
    {
        "January", "Feburary", "March", "April",
        "May", "June", "July", "August",
        "September", "October", "November", "December"
    };

    return 0;
}
```



In-Class Exercise 7-1

Write a program that takes a word less than 25 characters long and print a statement like this:

```
Input a word: fractal
fractal starts with the letter f.
```

8

String library

- `<string.h>` has functions for manipulating null-terminated strings.

- `int strlen(const char *s):` returns length of s
- `char *strcpy(char *s1, const char *s2):` copies s2 into s1 (Check if s1 has enough space.)
- `int strcmp(const char *s1, const char *s2)`: compares s1 and s2 (return 0 if they are the same, !=0 if they are different)
- `char *strcat(char *s1, const char *s2)`: appends s2 to s1 (Check if s1 has enough space.)

9

Example – strlen()

```
#include <stdio.h>
#include <string.h>

int main()
{
    int t;
    t = strlen("Czech Republic"); // 14 not counting '\0'
    printf("%d",t);
    return 0;
}
```

10

Example – strcmp()

```
#include <stdio.h>
#include <string.h>
int main()
{
   char szKey[] = "apple";
   char szInput[80];
   for (;;)
   {
       printf("Guess my favorite fruit?\n");
       scanf(“%s”, szInput);
       if (strcmp (szKey,szInput) == 0) break;
   }
   printf("Correct answer!");
   return 0;
}
```

11

Example – strcpy()

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[]="Sample string";
   char str2[40];
   char str3[40];
   strcpy (str2,str1);
   strcpy (str3,"copy successful");
   printf ("str1: %s\nstr2: %s\nstr3: %s\n",
            str1,str2,str3);
   return 0;
}
```

12

Example – strcat()

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str[80];
   strcpy (str,"these ");
   strcat (str,"strings ");
   strcat (str,"are ");
   strcat (str,"concatenated.");
   printf("%s", str);
   return 0;
}
```

13

In-Class Exercise 7-2

Write the `int strlen(const char *s)` function. This function takes a string and returns the number of characters, not counting the terminating null.

```
#include <stdio.h>

int strlen(const char* s)
{
    /* your code goes here */
}

int main()
{
    char* p = "hello, world";
    printf("%d\n", strlen(p));
    printf("%d\n", strlen(p + 7));
    return 0;
}
```

14

More String library functions

- `char* strncpy(char *s1, const char *s2, size_t n)`: copies the first n characters in s2 into s1
- `int strncmp(const char* s1, const char* s2, size_t n)`: compares the first n characters in s1 and s2 (return 0 if they are the same, !=0 if they are different)
- `char* strncat(char *s1, const char *s2, size_t n)`: appends the first n characters in s2 to s1

15

Example – strncmp()

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[3][5] = { "R2D2" , "C3PO" , "R2A6" };
    int n;
    puts ("Looking for R2 astromech droids...");
    for (n=0 ; n<3 ; n++)
        if (strncmp(str[n],"R2xx",2) == 0)
        {
            printf("found %s\n",str[n]);
        }
    return 0;
}
```

16

Global (External) vs. Local Variables

- A local variable is declared inside a function body.
  - It is local to a function, i.e., it only exists within this function.

```
int sum_digits(int n) {
    // local variable
    int sum = 0;
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}

void print_sum(int sum) {
    printf("sum is %d\n", sum);
}
```

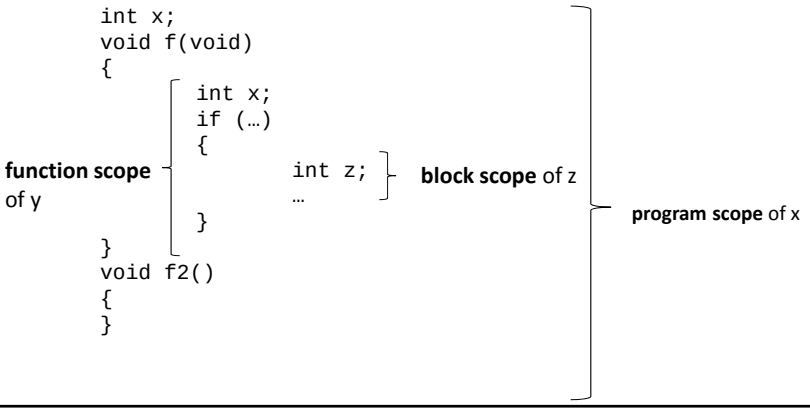
- A global variable is declared outside of any function.
  - It can be accessed anywhere in the program

```
int sum; // global variable
void sum_digits(int n) {
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }
}

void print_sum() {
    printf("sum is %d\n", sum);
}
```

17

Variable Scope

- The scope of a variable is the region within a program where it is defined and can be used.
  - program scope (global var); function/block scope (local var)



18

Variable Scope Example

```
int i;

void f(int i)
{
        i = 1;
}

void g()
{
        int i=2;
        if (i > 0) {
                int i;
                i = 3;
        }
        i = 4;
}

void h()
{
        i = 5;
}
```

19

Pros & Cons of Global Variables

Why not declare all variables global?

- External variables are very convenient when functions (e.g., `sum_digits() & print_sum())` must share a variable.
  - Passing variables between functions involves writing more code.
- What are the tradeoffs for global vs. local variables?
  - Say if you need to change the type of a global variable (must check all functions that use this variable)
  - Say if you need to debug a glob al variable that has a wrong value (which one is the guilty function?)
  - Say if you want to reuse a function (in another program) that rely on global variables

20

In-Class Exercise 7-3

Write a simple game-playing program using a global variable. The program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible. Your program should have the following input & output:

```
Guess the secret number between 1 and 100
Enter guess: 55
Too low, try again.
Enter guess: 65
Too high, try again.
Enter guess: 61
You won in 3 guesses!
Play again? (Y/N) Y

Guess the secret between 1 and 100
Enter guess: 31
```

21

More on Exercise 7-3

You program should have the following global variable and functions.

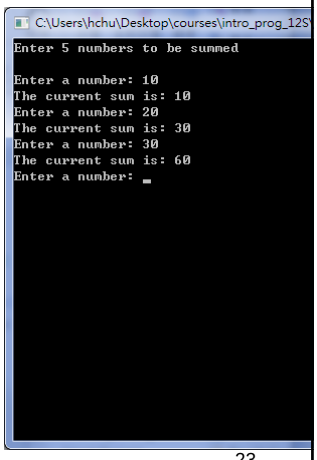
```
#include <stdio.h>
#include <time.h>

int secret_num; // global var to store the secret num
void init_num_generator()
{
        srand(time(NULL)); // init the seed of random number
generator
}

/* select a new secret number */
void new_secret_num()
{
        secret_num = rand() % 100 + 1; // a random number 0-99
}

/* continuously read user guesses and tell too low/high/correct */
void read_guesses();
```

22

Static Local Variables

A permanent storage inside a function so its value is retained throughout the program execution (vs. local variable, its storage is gone at the end of the function)

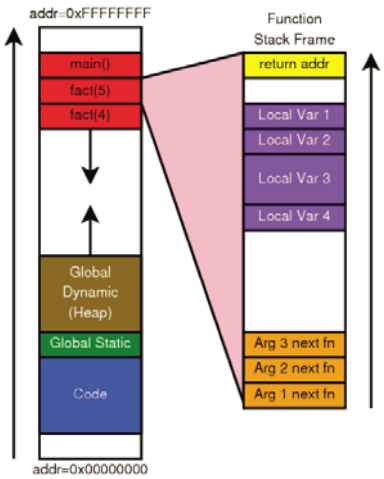
```
void sumIt(void)
{
    static int sum = 0;
    int num;
    printf("\nEnter a number: ");
    scanf("%d", &num);
    sum+=num;
    printf("The current sum is: %d",sum);
}

int main()
{
    int i =0;
    printf("Enter 5 numbers to be summed\n");
    for(i = 0; i<5; ++i)
        sumIt();
    printf("Program completed\n");
    return 0;
}
```



23

C Memory Model

- Program code
- Function variables
  - Arguments
  - Local variables
  - Return location
- Global Variables
  - Statically allocated
  - Dynamically allocated




```
int fact (int n)
{
   return(n*fact(n-1));
}
```

```
void main() { … fact(5); …}
```

24

The Stack

- Stores
  - Function local variables
  - Temporary variables
  - Arguments for next function call
  - Where to return when function ends

25

The Stack

- Managed by compiler
  - One stack frame each time function called
  - Created when function called
  - Stacked on top (under) one another
  - Destroyed at function exit

26

What can go wrong?

- Recall that local variables are stored on the stack
- Memory for local variables is deallocated when function returns
- Returning a pointer to a local variable is almost always a bug!

```
char *my_strcat(char *s1, char *s2)
{
        char s3[1024];
        strcpy(s3, s1);
        strcat(s3, s2);
        return s3;
}
```

27

What Can Go Wrong?

- Run out of stack space
- Unintentionally change values on the stack
  - In some other function's frame
  - Even return address from function
- Access memory even after frame is deallocated

28

The Heap

- C can use space in another part of memory: the heap
  - The heap is separate from the execution stack
  - Heap regions are not deallocated when a function returns
- The programmer requests storage space on the heap
  - C never puts variables on the heap automatically
  - But local variables might point to locations on the heap
  - Heap space must be explicitly allocated and deallocated by the programmer

29

malloc()

- Library function in <stdlib.h>
  - Stands for memory allocate
- Requests a memory region of a specied size
  - Syntax: `void *malloc(int size)`
  - `void *` is generic pointer type

30

Usage

```
int main()
{
   int *p = (int *) malloc(10 * sizeof(int));
   if (p == NULL)
   {
        // do cleanup
   }
   // do something
   free(p);
   return 0;
}
```

- Good to check the return value from `malloc()`
- Must explicitly free memory when no longer in use

31

What Can Go Wrong?

- Run out of heap space: malloc returns 0
- Unintentionally change other heap data
- Access memory after free'd
- free memory twice

32

Usage

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

int main()
{
   int *p = (int *) malloc(10 * sizeof(int));
   if (p == NULL)
   {
        // do cleanup
   }
   // do something
   if (p != NULL)
   {
        free(p);
        p = NULL;
   }
   return 0;
}
```

33

Multidimensional Array

- On the stack: `int a[10][20];`
- Initialization: `int a[][] = {{1, 2, 3}, {4, 5, 6}};`
- Accessing the array: `a[1][0]`
- On the heap

```
int **a = (int **) malloc(10 * sizeof(int *));
for (int i = 0; i < 10; ++i)
{
  a[i] = (int *) malloc(20 * sizeof(int));
}
```

- Don't forget to free them!

34

Exercise 7-4

Write a program that (1) asks for the number of friends, (2) asks for a name, and (3) checks a series of strings to see which one matches the names of friends. Please use

- Use `malloc()` to create this multi-dimensional array of friends' names. You can assume that the friends' names are less than 10 characters.
- Use `<string.h>` library
- Use `scanf("%s", ..)` to read a name separated by one or more whitespaces

```
Input the number of your friends: 4
John Polly Peter Jane
Input a name: Jane
Jane is friend #4
Input a name: Mary
Mary is not a friend
```

35