Principles of Computer Science II

Nadeem Abdul Hamid

CSC121A - Spring 2005

Lecture Slides 14 - Pointers, Functions, Storage Classes

Pointers

- Pointers are integer values that refer to addresses in memory
  - `scanf( "%d", &v );` — *"address of" operator*
- `int *` ==> "pointer to int" type

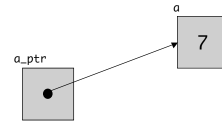
```
int i, *p; /* i: int,  p: pointer to int */
i = 4;
p = &i;    /* p contains memory address of i */
p = 0;
p = NULL;  /* defined in stdio.h to be 0 */
p = (int*) 1307;   /* absolute address in memory */
                   /* (very dangerous) */
```

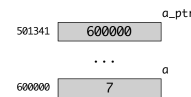

Naming Conventions

- Good practice to name pointers with "p" or "ptr" in the name
  - `y_ptr`
  - `yPtr`
  - `name_ptr`
- Asterisk (*) in declaring variables doesn't distribute to all names in a declaration:
  - `int *p, q;   /* not the same as: */`
  - `int *p, *q;`

Pointer Operations

```
int a = 7;
int *a_ptr = &a;
```





Pointer "Deferencing"

- Asterisk (*) is the *indirection* or *dereferencing* operator

```
int a = 7;
int *a_ptr = &a;

printf( "The value of a is %d", *a_ptr );
```

- Dereferencing a pointer that is not properly initialized/assigned a location in memory is error:
  - Fatal execution error
  - Accidentally modify other (important) data and program continues running with incorrect results or crashes the whole system later on

Pointer Operations (cont.)

```
int a = 7;
int *a_ptr = &a;

printf( "The address of a is %p (%u)"
        "\nThe value of a_ptr is %p (%u)", &a, &a, a_ptr, a_ptr );

printf( "\n\nThe value of a is %d"
        "\nThe value of *a_ptr is %d", a, *a_ptr );

printf( "\n\nShowing that * and & are complements of"
        "each other\n&*a_ptr = %p"
        "\n*&a_ptr = %p\n", &*a_ptr, *&a_ptr );
```

```
The address of a is 0xbffffc48 (3221224520)
The value of a_ptr is 0xbffffc48 (3221224520)

The value of a is 7
The value of *a_ptr is 7

Showing that * and & are complements ofeach other
&*a_ptr = 0xbffffc48
*&a_ptr = 0xbffffc48
```

Call-by-Reference

- Function arguments in C are passed strictly using "call-by-value" mechanism
  - *Values* of variables are copies into corresponding function parameters
- "Call-by-reference" can be simulated by pointer parameters

7

A swap Function

```
void swap( int, int );

int main(void) {
    int a = 5, b = 8;
    printf( "%d --- %d\n", a, b );
    swap( a, b );
    printf( "%d --- %d\n", a, b );

    return 0;
}

void swap( int a, int b ) {
    int t = a;
    a = b;
    b = t;
}
```

8

A Working swap Function

```
void swap( int *, int * );

int main(void) {
    int a = 5, b = 8;
    printf( "%d --- %d\n", a, b );
    swap( &a, &b );
    printf( "%d --- %d\n", a, b );

    return 0;
}

void swap( int *p, int *q ) {
    int t = *p;
    *p = *q;
    *q = t;
}
```

9

Void Pointers

- A pointer to void (`void *`) is a generic pointer that represents any pointer type
  - Any pointer type can be assigned to a void pointer and vice versa, without any need for a cast
  - Void pointers cannot be (directly) dereferenced (syntax error)

```
int a;
void *p = &a;
printf("a=%d\n", a);
*(int*)p = 5;
printf("a=%d\n", a);
```

10

- The ________ of an identifier is the part of the program in which the identifier is known or accessible.
  A. "Scope"
- Basic rule: identifiers are accessible only within the block in which declared.

```
#include <stdio.h>

int a = 10;
int *q = &a;

void foo( void );

int main( void ) {
    int a = 5;
    int *p = &a;
    printf( "%d\n", a );
    printf( "%d\n", *p );
    printf( "%d\n", *q );
    {
        int a = 7;
        printf( "%d\n", a );
        printf( "%d\n", *p );
    }
    printf( "%d\n", ++a );
    foo();
    return 0;
}

void foo( void ) {
    printf( "%d\n", a );
}
```

11

Storage Classes

- Section 8.6 (page 274-280)
- Every variable and function in C has two attributes:
  - Type
  - Storage class
- Four possible *storage classes*:
  - `auto` - default for variables declared in a block
  - `extern` - global variables with permanent storage
    - "Look for it elsewhere, either in this file or in some other"
    - All functions have external storage class
  - `register` - store variable in high-speed memory registers, if possible

12

External (Global) Variables

```
/* file1.c */

int a = 100;

int addone( int b ) {
    return b + 1;
}
```

```
/* file2.c */
#include <stdio.h>

int addone(int);

int main( void ) {
    extern int a;

    printf( "%d dalmations\n", addone(a) );
    return 0;
}
```

13

Storage Class `static`

- Two different uses:
  - Allows local variable to retain previous value when block is reentered
  - In connection with external declarations

```
void f( void ) {
    static int cnt = 0;
    int zero = 0;
    printf( "zero is now %d,"
            " count is now %d\n", ++zero, ++cnt );
}

int main( void ) {
    f();   f();   f();
    return 0;
}
```

14

Static External Variables

- Second (more subtle) use of `static`
  - Provides privacy mechanism for program modularity
  - Static external variables are scope restricted to the remainder of the source file in which they are declared
  - (Example - pseudorandom number generator - pg. 279-280)
- Note: Both external and `static` variables are initialized to zero automatically by C compiler, but not `auto` or `register` variables

15

`const` Qualifier

- Informs the compiler that the value of a particular variable should not be modified
  - (like `final` in Java)
- With pointers, four possible situations:
  - Non-constant pointer to non-constant data
    - Data can be modified, pointer can be changed
  - Non-constant pointer to constant data
    - Data cannot be modified, pointer can be changed
  - Constant pointer to non-constant data
    - …
  - Constant pointer to constant data
    - …

16

Pointers and `const`

Note: gcc compiler may ignore const qualifiers unless you compile with:

`gcc -pedantic-errors …`

```
int a       = 5;
const int b = 10;
int *p       = &a;
int * const q = &a;
const int *r  = &b;
const int * const s = &b;

printf("\n====\n"
       "a=%d, b=%d, p=%p,\n"
       "q=%p, r=%p, s=%p\n",
       a, b, p, q, r, s);

a++;
b++;     /* error */
(*p)++;
p++;
(*q)++;
q++;     /* error */
(*r)++;  /* error */
r++;
(*s)++;  /* error */
s++;     /* error */

printf("\na=%d, b=%d, p=%p,\n"
       "q=%p, r=%p, s=%p\n",
       a, b, p, q, r, s);
```

17

Passing Data Between Functions

```
int cubeA( int c ) { return c * c * c; }

void cubeB( int *c ) { *c = *c * *c * *c; }

int c;
void cubeC( void ) { c = c * c * c; }


int main( void ) {
    int a = 5;

    a = cubeA( a );
    printf( "%d\n", a );

    a = 5;
    cubeB( &a );
    printf( "%d\n", a );

    a = 5;
    c = a;
    cubeC();
    printf( "%d\n", c );

    return 0;
}
```

***Note:* Avoid global variables!!!**

18