Day12 A

Young W. Lim

2017-10-24 Tue

Outline

Based on

"C How to Program",
Paul Deitel and Harvey Deitel

Pointer Variable Definitions

- a pointer contains an address of another variable that contains a value
- pointers can be defined to point to objects of any type
  `int *p, char *q, float *r, ...`

int *p;	p can contain an address of an integer value
char *q;	q can contain an address of a character value
float *r;	r can contain an address of a float value
double *s;	s can contain an address of a double value

Indirection

- a (ordinary) variable name directly references a *value* . . . a
- a pointer variable name indirectly references a *value* . . . . *p

- referencing a value through a pointer is called indirection . . . *p

int *p;	*p refers to an integer value	*p : an integer variable
char *q;	*q refers to character value	*q : a character variable
float *r;	*r refers to a float value	*r : a float variable
double *s;	*s refers to a double value	*s : a double variable

Pointer Variable Initialization

- pointers should be initialized
  - either when they are defined (`int *p = &a;`)
  - or in an assignment statement (`p = &a;`)

- pointers can be initialized with
  - NULL : the pointer points to nothing (`q = NULL;`)
    a symbolic constant defined in <stddef.h>
  - 0 : the same as NULL, but NULL is preferred (`q = 0;`)
    the only integer that can be directly assigned to a pointer variable
    other integer requires a type cast
  - address of other variable (`p = &a;`)

Pointer Variable Assignment Examples

```
#include <stdio.h>

int main(void) {
  int i = 111;
  int *p = &i;
  unsigned long m;
  int *q;

  printf("sizeof(p) =%ld \n",
          sizeof(p));

  m = (unsigned long) p;

  printf("m= %lx \n",m);

  q = NULL;
  q = 0;
  q = (int *) m;

  printf("*q = %d \n", *q);

}
```

```
sizeof(p) =8
m= 7ffd767176cc
*q = 111
```

Pointer Operators

- & (address operator)
  - returns the address of its operand
  - its operand must be a variable (&a, &p)
- * (indirection operator)
  - returns the value of the object (*p)
    to which its operand points
- %p (conversion specifier)
  - in the `printf` statement prints
  - a memory location <u>address</u> as a <u>hexadecimal</u> integer

Pointer Arithmetic

- a limited set of arithmetic operations
    - ++ (increment)
    - -- (decrement)
    - +=, + integer addition
    - -=, - integer subtraction

Pointer Arithmetic : p+3

- `int *p;`
- `p + 3`

- the actual address of `(p+3)` :
  changes by 3 times the size of the object integer (`sizeof(int)`)
  to which the pointer `p` refers

- think `(p+3)` as the address of the element
  that are after 3 more integer elements
  from the current element to which `p` points

*p	*(p+1)	*(p+2)	*(p+3)
4-byte integer	4-byte integer	4-byte integer	4-byte integer

Pointer Arithmetic Examples (1)

	char *p	short *p	int *p
initial p	addr	addr	addr
p after ++p	addr + 1*1	addr + 1*2	addr + 1*4
p after p+=2	addr + 2*1	addr + 2*2	addr + 2*4
p after – –p	addr – 1*1	addr – 1*2	addr – 1*4
p after p–=2	addr – 2*1	addr – 2*2	addr – 2*4

- the content of a pointer variable `p` is the address `addr`

Pointer Arithmetic Examples (2)

```
#include <stdio.h>

int main(void) {
  char  a=-1, *p=&a;
  short b=-1, *q=&b;
  int   c=-1, *r=&c;


  printf("------------------\n");
  printf("p   = %p \n", p);
  printf("p+1 = %p \n", p+1);
  printf("p+2 = %p \n", p+2);

  printf("------------------\n");
  printf("q   = %p \n", q);
  printf("q+1 = %p \n", q+1);
  printf("q+2 = %p \n", q+2);

  printf("------------------\n");
  printf("r   = %p \n", r);
  printf("r+1 = %p \n", r+1);
  printf("r+2 = %p \n", r+2);

}
```

```
$ gcc -Wall pointer.c
$ ./a.out
-------------------
p   = 0x7ffdbc55bd89
p+1 = 0x7ffdbc55bd8a
p+2 = 0x7ffdbc55bd8b
-------------------
q   = 0x7ffdbc55bd8a
q+1 = 0x7ffdbc55bd8c
q+2 = 0x7ffdbc55bd8e
-------------------
r   = 0x7ffdbc55bd8c
r+1 = 0x7ffdbc55bd90
r+2 = 0x7ffdbc55bd94
```

Pointer Arithmetic Examples (3)

32-bit compilie

```
$ gcc -Wall -m32 pointer.c
$ ./a.out
-------------------
p   = 0xffd66289
p+1 = 0xffd6628a
p+2 = 0xffd6628b
-------------------
q   = 0xffd6628a
q+1 = 0xffd6628c
q+2 = 0xffd6628e
-------------------
r   = 0xffd6628c
r+1 = 0xffd66290
r+2 = 0xffd66294
```

default 64-bit compile

```
$ gcc -Wall pointer.c
$ ./a.out
-------------------
p   = 0x7ffdbc55bd89
p+1 = 0x7ffdbc55bd8a
p+2 = 0x7ffdbc55bd8b
-------------------
q   = 0x7ffdbc55bd8a
q+1 = 0x7ffdbc55bd8c
q+2 = 0x7ffdbc55bd8e
-------------------
r   = 0x7ffdbc55bd8c
r+1 = 0x7ffdbc55bd90
r+2 = 0x7ffdbc55bd94
```

Pointer Arithmetic Examples (4)

```
#include <stdio.h>

int main(void) {
  char  a=-1, *p=&a;
  short b=-1, *q=&b;
  int   c=-1, *r=&c;


  printf("-------------------\n");
  printf("p   = 0x%016lx \n", (unsigned long) p);
  printf("p+1 = 0x%016lx \n", (unsigned long) p+1);
  printf("p+2 = 0x%016lx \n", (unsigned long) p+2);

  printf("-------------------\n");
  printf("q   = 0x%016lx \n", (unsigned long) q);
  printf("q+1 = 0x%016lx \n", (unsigned long) q+1);
  printf("q+2 = 0x%016lx \n", (unsigned long) q+2);

  printf("-------------------\n");
  printf("r   = 0x%016lx \n", (unsigned long) r);
  printf("r+1 = 0x%016lx \n", (unsigned long) r+1);
  printf("r+2 = 0x%016lx \n", (unsigned long) r+2);

}
```

Pointer Arithmetic Examples (5)

```
$ gcc -Wall pointer.c
$ ./a.out
-------------------
p   = 0x00007ffdfc98e4e9
p+1 = 0x00007ffdfc98e4ea
p+2 = 0x00007ffdfc98e4eb
-------------------
q   = 0x00007ffdfc98e4ea
q+1 = 0x00007ffdfc98e4eb
q+2 = 0x00007ffdfc98e4ec
-------------------
r   = 0x00007ffdfc98e4ec
r+1 = 0x00007ffdfc98e4ed
r+2 = 0x00007ffdfc98e4ee
```

Simulating pass by reference

- all arguments are passed by value in C
- simulating pass by reference in C
  - using pointers and the indirection operator . . . . (`int *p`, `*`)
- to pass a variable by reference
  - use `& variable name`
  - to pass the address of the variable . . . . (`&a`)
- to receive the address arguement
  - define a pointer parameter variable . . . . (`int *p`)
- to modify the value of the variable within a function
  - use `* pointer parameter` . . . . (`*p=100;`)

Passing Pointers and Arrays

- the compile does not differentiate
    - a function receives a pointer
    - a function receives a single subscripted array, i.e. an 1-d array
- the programmer must make sure
    - a function receives an array (a set of elements)
    - a function receives a single variable passed by reference
- the compiler converts
    - `int b[]` into `int *b`
    - a single subscripted array parameter into a pointer parameter