

Computer Programming

Pointer Arithmetic. Function Pointers

Marius Minea

marius@cs.upt.ro

23 November 2015

Pointer do's and dont's (recap)

*p is NOT a pointer! unless p is `char **`, `int **`, etc.
p is the pointer. *p is the *object*/value at address p

Programs work with *data*.

Pointers are *addresses*, they only *point* to data.

Don't declare a pointer unless you have what it should point to.

except: dynamic allocation (provides pointer *and* data space

```
char *p = &s[i];      if array char s[40]; declared before  
char *p = "test";    data is constant string  
char *p = argv[0];   data put there by runtime system
```

Declare *data* and pass *address* for function to fill in data:

```
int n; if (scanf("%d", &n) == 1) ...  
char *end; double d = strtod(s, &end);  
int x, y; swap(&x, &y);
```

Arrays and pointers

The *name of an array* is a *constant address*

declaring an array allocates a memory block for its elements

the array's *name* is the *address* of that block (of first element)

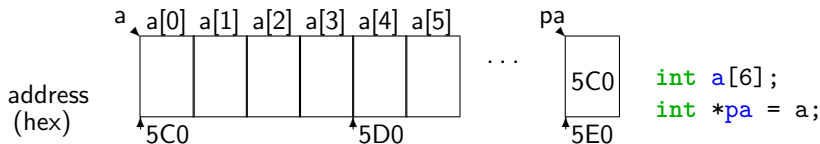
$\&a[0]$ is same as a and $a[0]$ is same as $*a$

Can declare `sometyp a[LEN], *pa;` and assign `pa = a;`

Similar: a and pa have same type: *sometyp**

But: pa is a *variable* \Rightarrow uses memory; *can assign* `pa = addr`

a is a *constant* (array has fixed address) *can't assign* ~~`a = addr`~~



$*a$ and $*pa$: indirections with different operations in machine code:

$*a$ references object from *constant* address (*direct* addressing)

$*pa$ must first get *value* of variable pa (an address), loading it from the *constant* address $\&pa$) *then* dereference it (*indirect* addressing)

Arrays and pointers (cont'd)

In function declarations, these are the same (first becomes second):
size_t strlen(char s[]); becomes size_t strlen(char *s);

As array declarations they are *different!*

Array: char s[] = "test"; s[0] is 't', s[4] is '\0' etc.

s is a *constant address* (char *), not a variable in memory

CANNOT assign s = ... but may assign s[0] = 'f'

sizeof(s) is 5 * sizeof(char) &s is s (but different type)

but with different type, address of 5-char array: char (*) [5]

sizeof (entire array) is not strlen (up to '\0')
--

Pointer: char *p = "test"; p[0] is 't', p[4] is '\0' (same)

p is a *variable of address type* (char *), has a memory location

CANNOT assign p[0] = 'f' ("test" is a string *constant*)

can do p = s; then p[0] = 'f'; can assign p = "ana";

sizeof(p) is sizeof(char *) &p is NOT p

⇒ WRONG: scanf("%4s", &p); RIGHT: scanf("%4s", p);

(if p is valid address and has room)

Pointer arithmetic

A variable v of a *sometype* takes up `sizeof(sometype)` bytes

\Rightarrow `&v + 1` is the address *after* the space allocated to v

numerically larger than `&v` by `sizeof(sometype)` bytes

+ on a pointer increments by an *object* (not a byte)

+ on a street address: advances by one house, not one meter

1. *Add/subtract* pointer and integer: like address of array element

`a + i` means `&a[i]` and `*(a + i)` means `a[i]` `3[a]` is `a[3]`

increment `++a`, `a++`: a becomes $a + 1$ before/after evaluation

```
// returns pointer to end of s; stops at null character '\0'
```

```
char *endptr(char *s) { while (*s) ++s; return s; }
```

2. *Difference*: only for pointers of *same* type (and in same array!)

= number of objects of *type* that fit between the two addresses

To get the number of bytes, convert pointers to `char *` (type cast):

```
p - q == ((char *)p - (char *)q) / sizeof(type)
```

No other arithmetic operations between pointers are defined!

May use comparison operators (`==`, `!=`, `<`, etc.)

Pointer arithmetic (cont.)

<code>pointer + int = pointer</code> (of same type)

Pointer arithmetic is only valid *within* allocated objects
exception: can take address *just* beyond (at end) of array
`int a[LEN], *end = a + LEN;`

In standard: `a+LEN+1` is *not* a valid address (beyond legal memory)
In practice: runtime won't protect from overflow; think carefully!

Arithmetic is not defined on `void *`. Cast to `char *` to add `int`

Pointer arithmetic and operator precedence

`++` (and `--`) have higher precedence than `*` (indirection)

`*p++` `++` applies to `p`: take value, (post)increment *pointer*

`(*p)++` (post)increments the *value* at address `p`

`*++p` takes value after incrementing pointer

`++*p` increments value at pointer (expression has that value)

Pointers and indices

same meaning: “to indicate” = “to point to”

To write `a[i]`, need two variables and one addition (base + offset) and multiplication with size of type (if not 1)

Simpler: directly with pointer to element `&a[i]` (`a+i`)
increment pointer rather than index when traversing array

```
char *strchr_i(const char *s, int c) { // search char in s
    for (int i = 0; s[i]; ++i) // traverse string up to '\0'
        if (s[i] == c) return s + i; // found: return address
    return NULL; // not found
}
```

```
char *strchr_p(const char *s, int c) {
    for (; *s; ++s) // use parameter for traversal
        if (*s == c) return s; // s points to current char
    return NULL; // not found
}
```

Pointers and indices (cont'd)

```
char *strcat_i(char *dest, const char *src)
{
    int i = 0, j;
    while (dest[i]) ++i;
    for (j = 0; src[j]; ++j)
        dest[i+j] = src[j];
    dest[i+j] = '\0';
    return dest;
}

char *strcat_p(char *dest, const char *src)
{
    char *d = dest;      // need to save dest for return
    while (*d) ++d;
    while (*d++ = *src++);
    return dest;
}
```


Pointers and multidimensional arrays

A bidimensional array (matrix) is declared as `type a[DIM1][DIM2];`
`a[i]` is address (const `type *`) of an array (line) of DIM2 elements
`a[i][j]` is j^{th} element in array `a[i]` of DIM2 elements
`&a[i][j]` or `a[i]+j` is DIM2*i+j elements after address `a`
⇒ a function with array parameter needs all dimensions except first
⇒ must declare as `functype f(etype t[][DIM2]);`

```
char t[12][4]={"jan",..., "dec"}; char *p[12]={"jan",..., "dec"};
```

`t` is matrix (2-D char array)

j	a	n	\0
f	e	b	\0
...			
d	e	c	\0

`t` uses 12 * 4 bytes

`t[6] = ...` is WRONG

`t[6]` is constant address of line 7

can do `str(n)cpy(t[6], ...)`

`p` is array of pointers

0x460	→	j	a	n	\0
0x5C4	→	f	e	b	\0
...					
0x9FC	→	d	e	c	\0

`p` uses 12*`sizeof(char *)` bytes

(+ 12*4 bytes for the string constants)

`p[6]="july"` changes an address

(element 7 from pointer array `p`)

Indices or pointers: use sensibly

Declare index in `for` loop header whenever possible (since C99)

enforces scope, visually clear, avoids affecting other loops

Do use indices if more suggestive, though combinations are possible

```
void matmul_i(unsigned m, unsigned n, unsigned p, double a[m][n],
              double b[n][p], double c[m][p]) {
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < p; ++j) {
            c[i][j] = 0;
            for (int k = 0; k < n; ++k) c[i][j] += a[i][k]*b[k][j];
        }
}
```

```
void matmul_p(unsigned m, unsigned n, unsigned p, double a[m][n],
              double b[n][p], double c[m][p]) {
    double *lastl = a[m];
    for (double *lp = a[0], *dp = c[0]; lp < lastl; lp += n)
        for (int j = 0; j < p; ++j, ++dp) {
            *dp = 0;
            for (int k = 0; k < n; ++k) *dp += lp[k]*b[k][j];
        }
}
```

// could you use more pointers ? For b perhaps ?

Type casts, void * and typedef

void * is used for addresses of any/unspecified type
⇒ *can't dereference* a **void *** (don't know what it points to)
but can assign to/from pointer of any other type
any pointer OK as arg/result for function declared with **void ***

Type cast is a unary *operator*, written as *(type-name) expression*
the value of *expression* is converted to the type *type-name*

convert int to real **(double)**sum/cnt //force real division
dereference a **void *** ***(char *)**p //char at address p

typedef is a keyword used to define a *new name* for type

Syntax: **typedef** *declaration* the identifier becomes a type *name*

```
typedef uint16_t u16; // u16 is synonym for type uint16_t  
typedef char line[80]; //line: type for array of 80 chars  
line text[100]; //text is array of 100 lines
```

Function pointers

A function *name* is its *address* (a pointer) – like for arrays

We can *declare* pointers of function type. Compare:

`int f(void);` declares a *function* returning int

`int (*p)(void);` declares *pointer to function* returning int

declare *function*: `restype fct (type1, ..., typeN);`

declare *function pointer*: `restype (*pfct) (type1, ..., typeN);`

Can assign `pfct = fct` with the name of an existing function

CAUTION! Need parantheses for `(*pointer)`, otherwise:

`int *fct(void);` is a function returning *pointer to int*

Function name is pointer \Rightarrow can call function using pointer

```
#include <math.h>
void printvals(double (*f)(double)) { // function parameter
    for (int i=0; i<10; ++i) printf("%f\n", f(.1*i));
}
int main(void) { printvals(sin); printvals(cos); return 0; }
```

Using function pointers

stdlib.h: binary search for key in sorted array; and quicksort

```
void *bsearch(const void *key, const void *base, size_t nmem,  
             size_t size, int (*compar)(const void *, const void *));  
void qsort(void *base, size_t num, size_t size,  
           int (*compar)(const void *, const void *));
```

address of array to sort, element count and size

address of comparison function, returns int <, = or > 0)

has void * arguments, compatible with pointers of any type

```
typedef int (*comp_t)(const void *, const void *); // cmp fun  
int intcmp(int *p1, int *p2) { return *p1 - *p2; }  
int tab[5] = { -6, 3, 2, -4, 0 }; // array to sort  
qsort(tab, 5, sizeof(int), (comp_t)intcmp); // sort ascending
```

Can also declare function with void *, do cast in function

```
int intcmp(const void *p1, const void *p2)  
    { return *(int *)p1 - *(int *)p2; }  
qsort(tab, 5, sizeof(int), intcmp); // no cast, has right type
```

When to use pointers ?

When the language *forces* us to:

arrays (memory blocks) cannot be passed / returned from functions
only their *address* (array name is its address)

addresses carry *no size* information \Rightarrow must pass size parameter

strings: a string (constant or not) is a `char *`
need not pass size, since null-terminated

functions: a function name is its address

When a function needs to modify variable passed from outside
pass *address* of variable

WARNING! Any address passed to a function needs to be valid
(point to allocated memory)

functions *use* their arguments \Rightarrow pointers must be valid