Computer Programming

Pointers

Marius Minea
marius@cs.upt.ro

12 November 2013

Pointers are addresses

Any lvalue (variable `x`, array element, structure field) of type `T` has an *address* `&x` of type `T *` where its value is stored.

An address is a numeric value, but is not an `int` / `unsigned` . It may be printed with format specifier `"%p"` in `printf`

Valid addresses are non-null. `NULL` indicates an invalid address

`(void *)0`   0 cast to type `void *`

We need to know how to

1. *declare* a variabile of pointer (address) type
2. *obtain* a pointer (address) value
3. *use* a pointer (address) value

To use pointers correctly, need to (like for all variables/values):

1. be aware of their *type*
2. use the right *operators* / functions

Declaring, initializing and assigning pointers

Declaring pointers: *type* **ptrvar*;

⇒ the variable *ptrvar* may contain the address of a value of *type*

Examples: `char *s; int *p;`

When declaring several pointers, need * for *each* of them:

`int *p, *q;` two integer pointers

`int *p, q;` one pointer `p` and one integer `q`

Obtaining pointers

An array name is a pointer: `int tab[10], *a = tab;`

same as: `int tab[10]; int *a; a = tab;`

In `T tab[10];` array name `tab` has type `T *`

The address operator & yields a pointer: `int n, *p = &n;`

or: `int n; int *p; p = &n;`

A *string constant* has type pointer: `char *s = "test";`

same as: `char *s; s = "test";`

Derefencing a pointer

The *dereferencing (indirection)* operator `*` prefix operator

operand: pointer;

result: *object* (variable) indicated by pointer

`*p` is an *lvalue* (can be assigned, like a variable)
can also be used in an expression, like any value of that type

The `*` operator is the *inverse* of `&`:

`*&x` is the object at address `&x`, thus `x`

`&*p` is the address of the value at address `p`, thus `p`

```
int x, y, *p = &x; y = *p; /* y = x */ *p = y; //x = y
```

`&` and `*` have *opposite effect on types*

`x` has type `T` ⇒ `&x` has type `T *`
`p` has type `T *` ⇒ `*p` has type `T`

Declaration and indirection

declaration $T * p$; may be read:

`T * p;`	p has type $T *$
`T *p;`	*p has type T
`char **s;`	address of char addr
`char *t[8];`	array of 8 char addr

Variable	Value	Address
`int x = 5;`	5	0x408
	...	
`int *p=&x;`	0x408	0x51C
	...	
`int **p2=&p;`	0x51C	0x9D0

WARNING: A *declaration* with *initializer* is NOT an *assignment* !

`int t[2] = { 3, 5 };` initializes `t`. WRONG: ~~`t[2] = { 3, 5 };`~~

`int x, *p = &x;` is like `int x; int *p; p = &x;`
(p is initialized/assigned, NOT *p). ~~`*p = &x`~~ is a type error!

`char *p = "sir";` is `char *p; p = "sir";` WRONG: ~~*p = "sir";~~

The * in declarations is NOT an indirection operator!
* is written next to the declared variable, but belongs to the *type*!

Using pointer parameters: assignment in functions

A function CANNOT change a variable passed as parameter
because the *value* is passed, not the variable itself

But, with a variable's *address* `p`, we may *use* its value: `...= *p;`
assign it: `*p =...;`

Having a variable's *address*, a function may *write* to it (e.g. `scanf`).

```
void swap (int *pa, int *pb) { // swaps values at 2 addresses
  int tmp; // keeps first changed value
  tmp = *pa; *pa = *pb; *pb = tmp; // integer assignments
}
```

Ex.: `int x = 3, y = 5; swap(&x, &y); // now x = 5, y = 3`

We use *addresses as function parameters*:
to pass *arrays* (can't pass array *contents* in C)
to return *several values* (`return` allows only one)
e.g. min *and* max of an array; result *and* error code

ERROR: no initialization

It's an *ERROR* to use *any uninitialized variable*

~~`int sum; for (i=0; i++ < 10; ) sum += a[i];`~~ `// initially??`

⇒ program behavior is *undefined* (best case: random initial value)

Pointers must be initialized before use, like any variables

with the *address* of a variable (or another initalized pointer)

with a *dynamically allocated* address (later)

ERROR: ~~`int *p; *p = 0;`~~ *ERROR*: ~~`char *p; scanf("%20s", p);`~~

p is *uninitialized* (best case NULL, if global variable)

⇒ value will be written to unknown memory address

⇒ memory corruption, security vulnerability; program crash is luckiest case!

WARNING: a pointer is not an int. WRONG: ~~`int *p = 640;`~~ !

Address space is determined by system, not user

⇒ *CANNOT choose* an arbitrary address we want

A pointer is like a post-it note
  declare = get fresh one
  can write an address on it
  but initially there is none

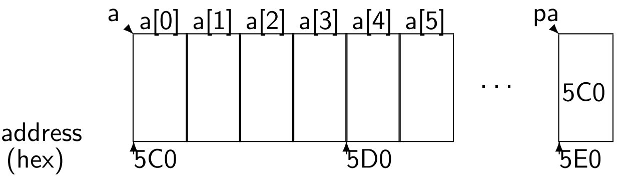
A variable is like a building
  has an address
  address fits on post-it
  building does not fit
  address not enough to build, need memory space

Programs process *data*, addresses are just helpers
  ⇒need actual data (vars, arrays) to get addresses from

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)
By declaring *type* `a[LEN], *pa;` we may assign `pa = a;`
`&a[0]` is equivalent with `a` and `a[0]` is equivalent with `*a`

Differences: address `a` is a *constant* (array has fixed address)
⇒ *we can't assign* `a =` *address*, but we may assign `pa =` *address*
`pa` is a *variable* ⇒ uses memory and has an address `&pa`




```
int a[6];
int *pa = a;
```

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

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 `p = s;` and 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);`

Pointer arithmetic

A variable `v` of a given `type` takes up `sizeof(type)` bytes
⇒ `&v + 1` is the address after the space allocated to `v`
numerically larger than `&v` by `sizeof(type)` bytes

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

```
char *endptr(char *s) { // returns pointer to end of s
  while (*s) ++s;      // stops at null character '\0'
  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.)

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 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 jth 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 *funtype* `f(`*eltype* `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

`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*)

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

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

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

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

(element 7 from pointer array `p`)

Command line arguments

command line: *program name* with *arguments* (options, files, etc.)

`gcc -Wall -o prog prog.c` `ls` *directory* `cp` *file1 file2*

`main` can access command line if declared with 2 args (*only* these):

`int argc` number of *words* in command line (arguments + 1)

`char *argv[]` array of argument addresses (strings)

```
#include <stdio.h>
int main(int argc, char *argv[]) { // or char **argv (same)
  printf("Program name: %s\n", argv[0]);
  if (argc == 1) puts("Program called with no arguments");
  else for (int i = 1; i < argc; i++)
    printf("Argument %d: %s\n", i, argv[i]);
  return 0;
}
```

`argv[0]` (first word) is program name, thus `argc >= 1`

array `argv[]` ends with a `NULL` element, `argv[argc]`

Run a command from program: `int system(const char *cmdline)`

returns -1 if can't run, or exit code of program

Formatted string reading/writing/conversions

Variants of `printf`/`scanf` with strings as source/destination

```
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

`sprintf` has *no limitation* ⇒ may overflow buffer. Use instead:

```
int snprintf(char *str, size_t size, const char *format, ...);
```

writing is limited to `size` chars including `\0` ⇒ safe option

Converting strings to numbers

```
int n; char *s;
if (sscanf(s, "%d", &n) == 1) ... //read correctly
```

(but we don't know where processing of string stopped)

```
long int strtol(const char *nptr, char **endptr, int base);
```

assigns to `*endptr` the address of first unprocessed char

```
char *end; long n = strtol(s, &end, 10);
```

base 10 or other

also `strtoul` for `unsigned long`, `strtod` for base 10 `double`

```
int n = atoi(s);
```

returns 0 on error, but also for `"0"`

use only when string known to be good

Function pointers

Sometimes we wish to call different functions in a program point
Example: array traversal with various kinds of processing

`for (int i = 0; i < len; ++i) f(&tab[i]);` various functions `f`

A function *name* is its *address*. Compare declarations:

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

We may assign `pfct = fct`; the name of a function is its address

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

CAUTION! Need parantheses around `(*pointer)`, otherwise:
`int *fct(void);` is a function returning *pointer to int*

Declare pointer type to make declarations of that type easier:
`typedef` in front of a declaration declares *type name*, not variable

`typedef void (*funptr)(void);` pointer to void function
`funptr funtab[10];` array of void function pointers

Using function pointers

Example: standard quicksort function `qsort` (`stdlib.h`)

```
void qsort(void *base, size_t num, size_t size,
                                    int (*compar)(void *, 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
```

Also: binary search for key in sorted array

```
void *bsearch(const void *key, const void *base, size_t nmemb,
       size_t size, int (*compar)(const void *, const void *));
```