Computer Programming

Pointers

Marius Minea
marius@cs.upt.ro

20 November 2017

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 array name is its address

A string is represented by its address, it is a `char *`

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

`NULL` is `(void *)0` i.e., 0 cast to type `void *`

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

For low-level systems programming:
Types `intptr_t` and `uintptr_t` (from `stdint.h`)
are the right size to hold a `void *`.

Pointers are used like everything else

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. *initialize* them correctly
3. use the right *operators* / functions

Declaring 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`

Initialize pointers in declarations wherever possible

like with any variable: don't risk using uninitialized values

Initialization and assignment

Obtaining pointer values:

From *an array name* (a pointer):

```
int tab[10], *a = tab;
```

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

Declaring `T tab[N];` array name `tab` has type `T *`

Taking *the address &* of a variable: `int n, *p = &n;`

same as: `int n; int *p; p = &n;`

A *string constant* is a pointer to the contents (to first char):

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

Dereferencing a pointer

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

`*p` gives the *object* located at address `p`

operand: pointer (address); 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

Declaration syntax suggests types!

`T *p;` says `T *` is the type of `p` and `T` is the type of `*p`

Address and dereference operators are opposites

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

`*&x` is the object at the address of `x`, that is, `x`

`&*p` is the address of the object at address `p`, that is, `p`

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

Always check the types!

`x` has type `T` $\Rightarrow$ `&x` has type `T *`

`p` has type `T *` $\Rightarrow$ `*p` has type `T`

We can have pointers to pointers to pointers ...

Any variable has an address ⇒ pointer variables have addresses
Any expression has a type:
The address of a variable of type `T` has type `T *`
The address of a variable of type `T*` has type `T **`   etc.

Having declared `int *p;` the type of &p is `int **`
⇒ we can declare `int **p2` and initialize/assign it with &p

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

Initialization and assignment are different!

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

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

Declaration `int *p;` suggests that `*p` is an `int`
but the variable declared is `p`, ~~NOT `*p`~~ (`*p` is not an identifier)
so the initializer is for `p`, ~~NOT for `*p`~~.

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

Pointers hold only addresses, not data!

Programs can't have just pointers. These must point to something (useful data: need variables to store it in).

Understand what each declaration means!

Declaring `int x;` means

I want to have an integer. What for? What value does it have?

⇒ Better: `int min = a[0]; //start with first element`

Declaring `char *p;` only means

I want to use the address of a char

DON'T KNOW WHAT ADDRESS. VARIABLE p UNINITIALIZED. NO CHARS DECLARED YET. NO ROOM TO STORE THEM.

Need:

`char *p = buf;` p points to array `char buf[10];` declared before

`char *p = "ana are mere";` p points to a *string constant*

`char *p = strchr(buf, '<');` returned by function, could be `NULL`

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 a *valid address* (of a variable), or an initialized 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

Using pointer parameters: assignment in functions

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

```
void nochange(int x) { ++x; printf("%d\n", x); }
void try(void) {
  int a = 5; nochange(a);     // nochange prints 6
  printf("%d\n", a);          // main still prints 5 !
}
```

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

Pointers as function parameters

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

Arrays as function parameters

When passing an array to a function, the *address is passed*

The name of the array represents its address

in `T` *tab*`[LEN];` the array name `tab` has type `T *`

`int f(int a[])` is same as `int f(int *a)`

Formatted processing/printing of strings

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[] = "-102 56 42";
if (sscanf(s, "%d", &n) == 1) ... //number OK
```

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

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

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

(if not needed, pass 2nd arg. `NULL`)

if base is 0, accepts octal/decimal/hex (as in C, like `%i` in scanf)

```
char *end; long n = strtol(s, &end, 10); //upto base 36
```

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

Command line arguments

command line: *program name* with *arguments* (options, files, etc.)
Examples: `gcc -Wall prog.c` or `ls` *directory* or `cp` *file1 file2*

`main` can access command line if declared with 2 args (*only* these):
`int argc` count of *words* in command line (1 + arguments)
`char *argv[]` arguments: array of strings, ends with `NULL`

```
#include <stdio.h>
int main(int argc, char *argv[]) {  // same as char **argv
  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;
} // run: ./a.out somestring anotherstring thirdstring etc
```

Run a command from program:

```
int system(const char *cmdline);
```

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

Declaring an array allocates a memory block for its elements

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

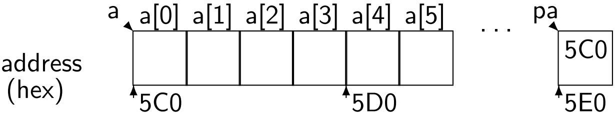
&a[0] is same as a   and   a[0] is same as *a

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

Similar: a and pa have same type: T*

But: pa is a *variable* ⇒ uses memory; *can assign* pa = *addr*

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




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

*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, loading it from &pa, *then* dereference it (*indirect* addressing)

Arrays and pointers (cont'd)

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 type is 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 assign `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

$\boxed{\text{pointer} + \text{int} = \text{pointer} \qquad \text{(of same type)}}$

A variable `v` of type `T` uses `sizeof(T)` bytes

⇒ `&v + 1` is the address *after* v's space (next object)

`&v + 1` is value of `&v` plus `sizeof(T)` bytes

+ on a pointer increments by *object size* (~~not one byte~~)

Pointer arithmetic: add

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

`a + i` means `&a[i]`

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

`a + i` means `i` *elements* past `a`, NOT ~~`i` bytes past `a`~~

for `char *a`   1 *element* = 1 *byte* ⇒ number also means bytes

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

Pointer arithmetic is only valid *within* the same array/object
exception: can take address *just* beyond (at end) of array

```
int a[LEN], *end = a + LEN;
```

`a+LEN+1` is *not* a valid address (beyond legal memory access)

WARNING! C has no overflow checks! Careful with indices!

Pointer arithmetic: difference

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

= number of objects of type `T` between the two addresses

```
&a[j] - &a[i] == j - i
```

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

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

No other arithmetic operations between pointers are defined!

May use comparison operators: `==`, `!=`, `<`, etc.

comparing order `<`, `<=` etc. only allowed within same structure

(relative memory placement of different objects is irrelevant)

No pointer arithmetic with `void *`

`void *` = pointer of unspecified type
don't know type of object ⇒ can't dereference, can't do arithmetic

But: `void *` are assignment-compatible with any pointer
Useful for writing functions that accept *any* pointer

Cast `void *` to `char *` to do arithmetic:

```
void setzero(void *a, unsigned cnt, unsigned size) {
  for (char *p = (char *)a + cnt * size; --p >= a; ) *p = '\0';
}
```

Pointer arithmetic and operator precedence

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

Increment pointer

`*p++`   `++` applies to `p`: take value, (post)increment pointer
value is object *at* original pointer value

`*++p`   increments pointer, then dereferences
value is next object *after* original pointer value

Increment value at pointer

`(*p)++`   (post)increments the value at address `p`
expression has the value *before* increment

`++*p`   (pre)increments value at address `p`
expression has the value *after* increment

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 char, of size 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++);      // string copy
  return dest;
}
```

Pointers and multidimensional arrays

A bidimensional array (matrix) is declared as *type* `a[DIM1][DIM2];`
for instance `int a[DIM1][DIM2];`

`a[i]` is constant address (`int *`) of an array of `DIM2` elements
(line of the matrix)

`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`

⇒ function with array parameter needs all dimensions except first

⇒ must declare as *sometype* `f(int t[][DIM2]);`

`a[i]` which is `*(a+i)` means `i` lines (×`DIM2` elements) after `a[0]`

⇒ `a` has type `int (*)[DIM2]` (pointer to array of `DIM2` ints)

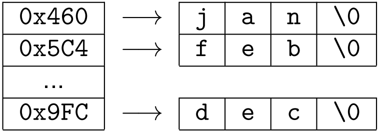
Matrix vs. array of pointers

`char t[12][4]={"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

`char *p[12]={"jan",...,"dec";}`
p is array of pointers



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 `strcpy(t[6], ...)` or `strncpy`

`p[6]="july"` changes an *address*
(element 7 from pointer array `p`)

Indices or pointers: use sensibly

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

- enforces scope, visually clear, avoids affecting other loops

Use whatever results in simpler, understandable code

```
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]) {
  for (double *lp = a[0], *dp=c[0], *end = a[m]; lp<end; 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];
    }
}
```

Type casts and typedef

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`

read bits of float as an int: `*(uint32_t *)&f`

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

Syntax: `typedef` *declaration*
the identifier that would have been a *variable* in the declaration becomes a *type name*

```
typedef uint16_t u16; // u16 is synonym for type uint16_t
// with just:  uint16_t u16;     it would be a variable
typedef char line[80]; //line: type for array of 80 chars
// with just:  char line[80];   it would be an array
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);` declares a function returning *pointer to int*
Function name is pointer ⇒ can call function using pointer

```
#include <math.h> // Example: f is a function parameter
void printvals(double (*f)(double)) { // arg.of f not named
  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 nmemb,
      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 ⇒ 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 ⇒ pointers must be valid