Computer Programming

Pointers

Marius Minea

marius@cs.upt.ro

14 November 2016

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, initializing and assigning pointers

Declaring pointers: *type* **ptrvar*;

$\Rightarrow$ 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

From *an array name* (a pointer): `int tab[10], *a = tab;`

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

Declaring `T tab[10];` 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 type of `p` and `T` is type of `*p`

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` ⇒ `&x` has type `T *`

`p` has type `T *` ⇒ `*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 the *address* of a variable (or another 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 *`

restype `f(`*eltype* `a[])` is same as *restype* `f(`*eltype* `*a)`

Conversions from 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 *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

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

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

```
#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
```

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

`argv[]` array 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