C Primer part 2

CS 351: Systems Programming

Melanie Cornelius

Slides and course content obtained with permission
from Prof. Michael Lee, <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

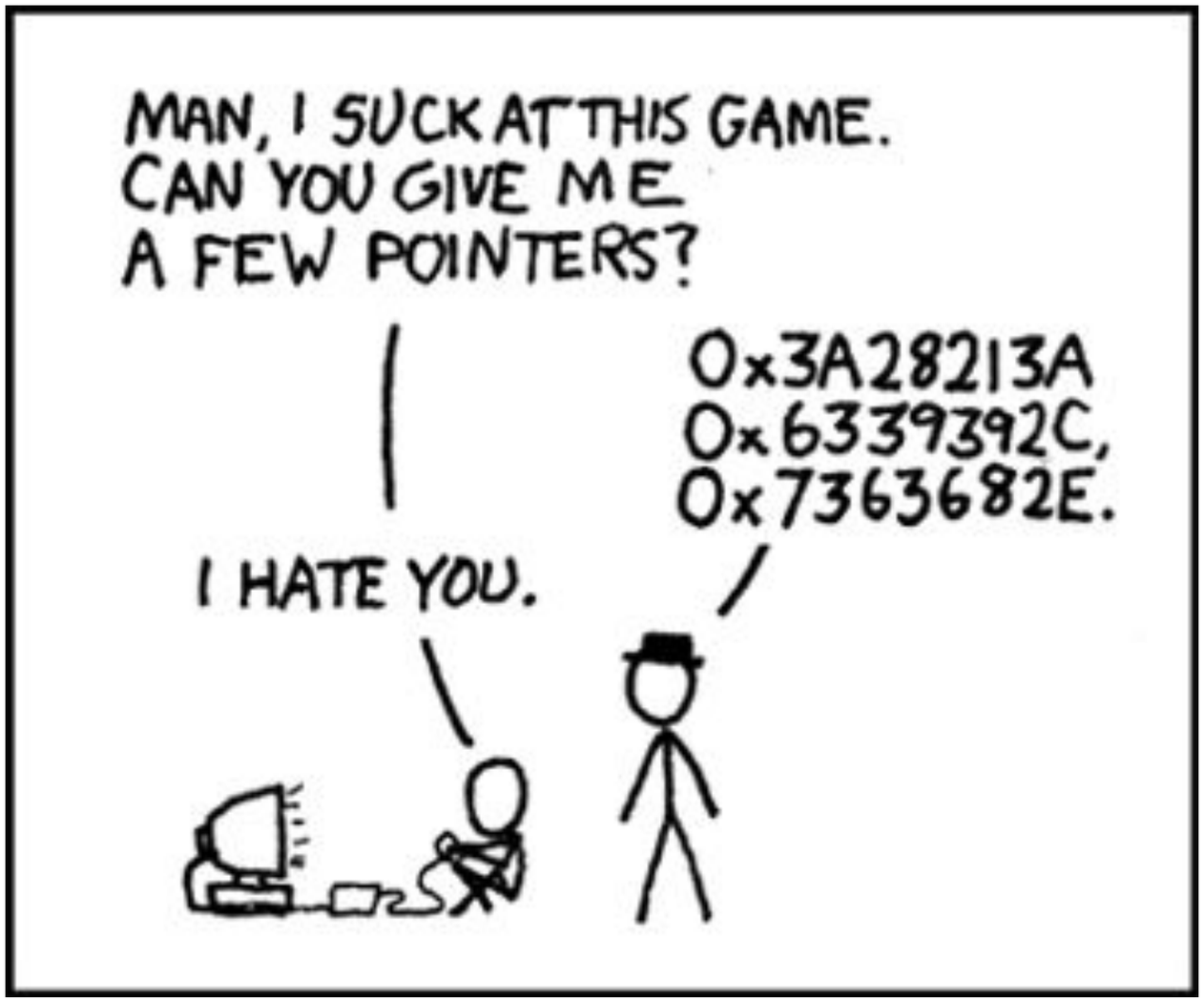
Science

Computer Science

Pointers

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

(don’t panic!)



a *pointer* is a variable declared to store a *memory address*

what's a *memory address*?

- an address that can refer to a datum in memory
- given address size w, range = 0 to 2^w-1
- width determined by machine *word size*
  - e.g., 32-bit machine → 32-bit address

e.g., for word size = 32, the following are valid memory addresses:

- 0
- 100
- 0xABCD1234
- 0xFFFFFFFF

i.e., an address is *just a number*

Q: by examining a variable's contents, can we tell if the variable is a pointer?

0x0040B100

No!

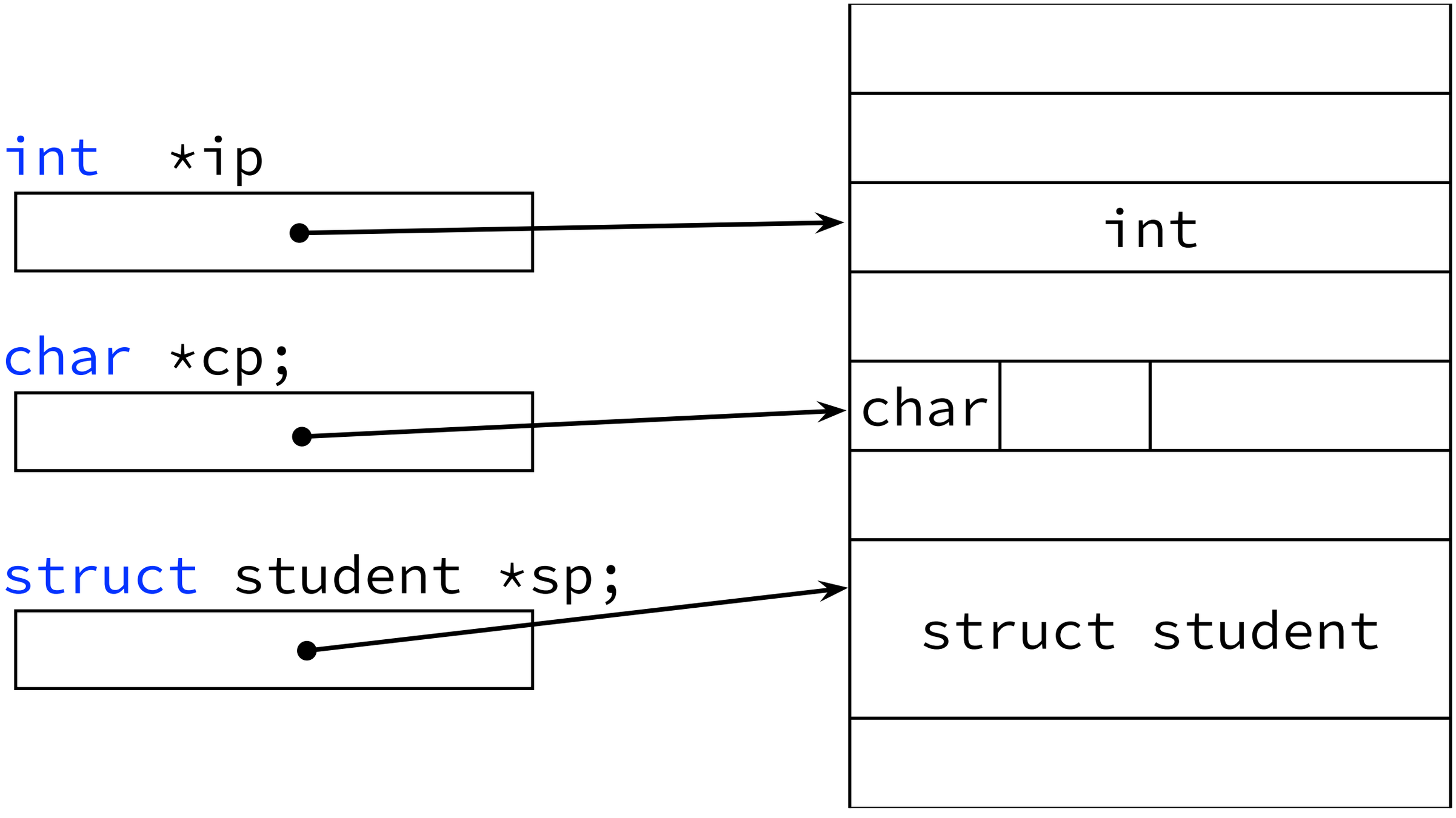
-a pointer is designated by its *static (declared) type*, not its contents

A pointer declaration also tells us the *type of data to which it should point*

declaration syntax:

```
type *var_name
```

(ex 1)


int *ip
char *cp;
struct student *sp;
int
char
struct student

Important pointer-related operators:

& : address-of (reference)

* : value-at (dereference)

```c
int i = 5;   /* i is an int containing 5 */
int *p;      /* p is a pointer to an int */

p = &i;      /* store the address of i in p */
```

```c
int j;       /* j is an uninitialized int */
j = *p;      /* store the value p points to into j*/
```

(ex 2)

```
int i, j, *p, *q;
i = 10;
```

Address	Data	
1000	10	(i)
1004	?	(j)
1008	?	(p)
1012	?	(q)

```
p = &j;
```

Address	Data	
1000	10	(i)
1004	?	(j,*p)
1008	1004	(p)
1012	?	(q)

```
q = p;
```

Address	Data	
1000	10	(i)
1004	?	(j,*p,*q)
1008	1004	(p)
1012	1004	(q)

```
*q = i;
```

Address	Data	
1000	10	(i)
1004	10	(j,*p,*q)
1008	1004	(p)
1012	1004	(q)

```
*p = *q * 2;
```

Address	Data	
1000	10	(i)
1004	20	(j,*p,*q)
1008	1004	(p)
1012	1004	(q)

```c
int main() {
    int i, j, *p, *q;

    i = 10;
    p = &j;
    q = p;
    *q = i;
    *p = *q * 2;
    printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);
    return 0;
}
```

```
$ gcc pointers.c
$ ./a.out
i=10, j=20, *p=20, *q=20
```

why have pointers?

Pass by value!

```
int main() {
    int a = 5, b = 10;
    swap(a, b);
    /* want a == 10, b == 5 */
    ...
}

void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

```
int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    /* want a == 10, b == 5 */
    ...
}
```

```
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}
```

(ex 3)

pointers enable *action at a distance*

action at a distance is an *anti-pattern*

i.e., an oft used but typically crappy programming solution

Swapping pointers?

```
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a, b, *c = &a, *d = &b;
    swap(&c, &d);
    /* want c to point to b, d to a */
}
```

```
$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:10: warning: passing argument 1 of ‘swap’ from
incompatible pointer type
pointers.c:10: warning: passing argument 2 of ‘swap’ from
incompatible pointer type
```

```
void swapp(int **p, int **q) {
    int *tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a, b, *c = &a, *d = &b;
    swapp(&c, &d);
    /* want c to point to b, d to a */
}
```

`int **` declares a
pointer to a pointer to an int

(ex 4)

Uninitialized pointers

- are like all other uninitialized variables
  - i.e., contain **garbage**
- dereferencing garbage ...
  - if lucky → crash
  - if unlucky → ???

“Null” pointers

- never returned by & operator
- safe to use as sentinel value (ex, before variable usefully populated)
- written as 0 in pointer context
  - for convenience, #define’d as NULL

Arrays and Arithmetic

Array:

contiguous, indexed region of memory

Declaration:

```
type arr_name[size]
```

- remember, declaration also allocates storage!

```c
int  i_arr[10];         /* array of 10 ints */
char c_arr[80];         /* array of 80 chars */
char td_arr[24][80];    /* 2-D array, 24 rows x 80 cols */
int  *ip_arr[10];       /* array of 10 pointers to ints */

/* dimension can be inferred if initialized when declaring */
short grades[] = { 75, 90, 85, 100 };

/* can only omit first dim, as partial initialization is ok */
int sparse[][10] = { { 5, 3, 2 },
                     { 8, 10 },
                     { 2 } };

/* if partially initialized, remaining components are 0 */
int zeros[1000] = { 0 };

/* can also use designated initializers for specific indices*/
int nifty[100] = { [0]  = 0,
                   [99] = 1000,
                   [49] = 250 };
```

(ex 5)

In C, arrays contain *no metadata*

i.e., ***no*** *implicit size*, ***no*** *bounds checking*

direct access to memory can be *dangerous*!

pointers ♥ arrays

- an array name is bound to the address of its first element
  - i.e., array name is a *const pointer*
- conversely, a pointer can be used as though it were an array name

(ex 6)

```c
int arr[100];
int *pa = arr;

pa[10] = 0;       /* set tenth element */

/* so it follows ... */

*(pa + 10) = 0;   /* set tenth element */

/* surprising! "adding" to a pointer
   accounts for element size -- does not
   blindly increment address */
```

```
int arr[100];
arr[10] = 0xDEADBEEF;

char *pa = (char *)arr;

pa[10] = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
DEADBEEF
```

```c
int arr[100];
arr[10] = 0xDEADBEEF;

char *pa = (char *)arr;

int offset = 10 * sizeof (int);

*(pa + offset) = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
DEADBE00
```

`sizeof`: an operator to get the size *in bytes*

- can be applied to a datum or type

```c
int arr[100];
arr[10] = 0xDEADBEEF;

char *pa = (char *)arr;

int offset = 10 * sizeof (int);

*(int *)(pa + offset) = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
0
```

strings are just 0 terminated char arrays

```
char str[]      = "hello!";
char *p         = "hi";
char tarr[][5] = {"max", "of", "four"};
char *sarr[]    = {"variable", "length", "strings"};
```

```
(gdb) x /7x str
0x7fffffffe1c0: 0x68    0x65    0x6c    0x6c    0x6f    0x21    0x00
(gdb) x /3x p
0x40062c:       0x68    0x69    0x00
(gdb) x /15x tarr
0x7fffffffe1b0: 0x6d    0x61    0x78    0x00    0x00    0x6f    0x66    0x00
0x7fffffffe1b8: 0x00    0x00    0x66    0x6f    0x75    0x72    0x00
(gdb) x /3a sarr
0x7fffffffe190: 0x40062f        0x400638            0x40063f
(gdb) x /s sarr[0]
0x40062f:       "variable"
```

Arrays of pointers and 2D arrays are very different

(ex 7)

```c
/* printing a string (painfully) */

int i;
char *str = "hello world!";
for (i = 0; str[i] != 0; i++) {
    printf("%c", str[i]);
}

/* or just */

printf("%s", str);
```

```c
/* Beware: */

int main() {
    char *str = "hello world!";
    str[12] = 10;
    printf("%s", str);
    return 0;
}
```

```
$ ./a.out
[1]    22432 segmentation fault (core dumped)  ./a.out
```

```c
/* the fleshed out "main" with command-line args */


int main(int argc, char *argv[]) {
    int i;
    for (i=0; i<argc; i++) {
        printf("%s", ((i < argc-1)? ", " : "\n") );
    }
    return 0;
}
```

```
$ ./a.out testing one two three
./a.out, testing, one, two, three
```