Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

1. C: pointers to memory
   - `scanf`, copying memory

Pointers: declaration

- All variables are represented by a sequence of bytes
  - `int`, `long` are interpreted as integer in two-complement
  - `float`, `double` are interpreted as floating point numbers according to the standard IEEE 754-1985
- A pointer variable is interpreted as **an address in memory**
- Declared by specifying the type of the variable it points to

  ```
  <type> * <identifier>;
  ```
- only the pointer is allocated, **not the variable it points to**!!
- Example

  ```
  int *pi1, *pi2, i, j;
  ```

  declares `pi1` and `pi2` as pointers to integer, `i` and `j` are just integers.
- Usually names of pointers contain “`p`” or “`prt`”

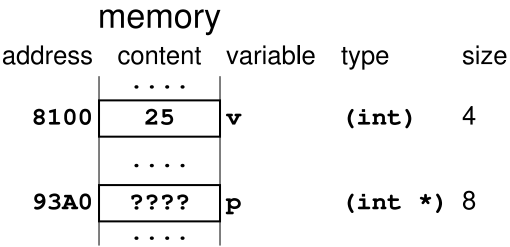
Pointers: example of usage

memory

address	content	variable	type	size
	....			
8100	**????**	**v**	**(int)**	4
	....			
93A0	**????**	**p**	**(int *)**	8
	....			

```
int v;
int * p;
```

Pointers: example of usage

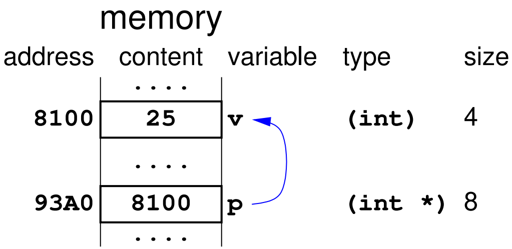


memory

address	content	variable	type	size
	....			
8100	**25**	**v**	**(int)**	4
	....			
93A0	**????**	**p**	**(int *)**	8
	....			

```
int v;
int * p;

v = 25;
```

Pointers: example of usage

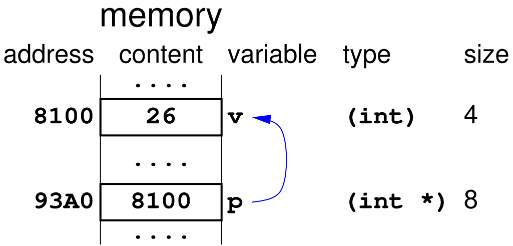


memory

address	content	variable	type	size
	....			
8100	25	v	(int)	4
	....			
93A0	8100	p	(int *)	8
	....			

```
int v;
int * p;

v = 25;
p = &v;  /* assignment of address of v to p */
```

Pointers: example of usage



memory

address	content	variable	type	size
	....			
8100	26	v	(int)	4
	....			
93A0	8100	p	(int *)	8
	....			

```
int v;
int * p;

v = 25;
p = &v;  /* assignment of address of v to p */
*p += 1; /* increment integer pointed by p */
printf("%d", v); /* what do we print? */
```

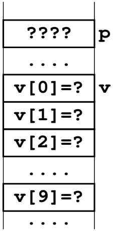
Operations with pointers

```
int v;
int * p;

v = 25;
p = &v;  /* &v is the "address of" the variable v */
*p += 1; /* *p is the variable "pointed" ("dereferenced") by p */
```

- **"address of"**: from a variable to its address in memory
  - The unary operator & can be **applied to any variable**
  - &v is the address in memory of the variable v
  - if v declared by <type> v, then &v is of type (<type> *)
- **dereferencing**: from the pointer to the variable it points to
  - The unary operator * can **only be applied to a pointer** (any variable p declared by <type> * p;)
  - If p is a pointer, *p is the variable pointed by p
  - **Warning**: "*" is used to both declare a pointer and to dereference it
- Can we write &p?

Arrays and pointers

- The array “variable” is a **constant** pointer to the first cell of the array
- If `<type> * p;`, then `p[i]` is the i-th element of an array of `<type>` starting at address `p`

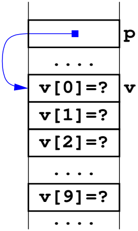


????
p
....
v[0]=?
v
v[1]=?
v[2]=?
....
v[9]=?
....

```
int v[10], *p;
```

Arrays and pointers

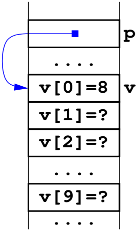
- The array “variable” is a **constant** pointer to the first cell of the array
- If `<type> * p;`, then `p[i]` is the i-th element of an array of `<type>` starting at address `p`



```
int v[10], *p;

p = v;      /* same as p = &v[0]; */
```

Arrays and pointers

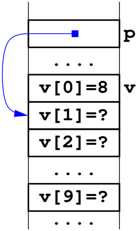
- The array "variable" is a **constant** pointer to the first cell of the array
- If `<type> * p;`, then `p[i]` is the i-th element of an array of `<type>` starting at address `p`



```
int v[10], *p;

p = v;        /* same as p = &v[0]; */
*v = 8;       /* same as v[0] = 8; */
```

Arrays and pointers

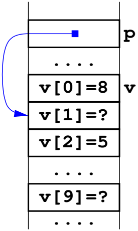
- The array "variable" is a **constant** pointer to the first cell of the array
- If `<type> * p;`, then `p[i]` is the i-th element of an array of `<type>` starting at address `p`



```
int v[10], *p;

p = v;      /* same as p = &v[0]; */
*v = 8;     /* same as v[0] = 8; */
p = &v[1];
```

Arrays and pointers

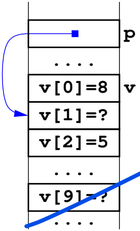
- The array "variable" is a **constant** pointer to the first cell of the array
- If `<type> * p;`, then `p[i]` is the i-th element of an array of `<type>` starting at address `p`



```
int v[10], *p;

p = v;       /* same as p = &v[0]; */
*v = 8;      /* same as v[0] = 8; */
p = &v[1];
p[1] = 5;    /* same as v[2] = 5; */
```

Arrays and pointers

- The array “variable” is a **constant** pointer to the first cell of the array
- If `<type> * p;`, then `p[i]` is the i-th element of an array of `<type>` starting at address `p`



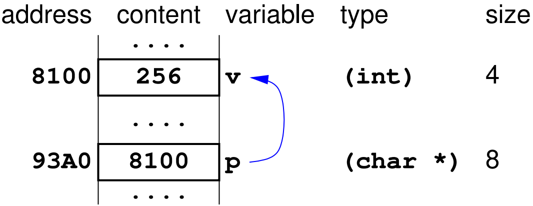
```
int v[10], *p;

p = v;       /* same as p = &v[0]; */
*v = 8;      /* same as v[0] = 8; */
p = &v[1];
p[1] = 5;    /* same as v[2] = 5; */
```

- the difference betwen a pointer `p` and an array `v` is that
  1. the name of arrays is **constant**, it cannot be assigned to a value

     ```
     v = &v[1];    /* ERROR */
     v = p;        /* ERROR */
     ```
  2. at declaration time
     - ★ `int v[10]` allocates a contiguous area to store 10 variables of type `int`
     - ★ `int * p` allocates a variable `p` to store only a pointer
  3. `sizeof(p)` is the size of the address `p`,
     `sizeof(v)` is the size of the array `v`

Casting a pointer to another type

- Addresses in memory always occupies the same number bytes
  - `sizeof(int *)` equals `sizeof(char *)` equals `sizof(double *)`
- Why do we care of the type of the pointer?
  - To properly interpret the pointed data
- By casting a pointer, it is changed the type of pointed data



address	content	variable	type	size
	....			
8100	256	v	(int)	4
	....			
93A0	8100	p	(char *)	8
	....			

```
int v = 256;
char * p;

p = (char *)&v;    /* &v casted from (int*) to (char*) */
*p = 1;            /* this is an assignment of a (char) */
printf("%d", v);   /* what do we print? */
```

- *test-ptr-cast.c*

Segmentation fault

- **Segmentation fault** is a **run time** error signaled by the operating system when the user attempts to read/write to some memory areas where the user has no right to access to

```
int *p;

v = *p;  /* Trying to read from unknown memory location.
          * It MAY trigger Segmentation Fault. */

*p = 5;  /* Trying to write to an unknown memory location.
          * It MAY trigger Segmentation Fault. */
```

- The following code tries to read and write everywhere
- *test-seg-fault.c*

Generic pointer (`void *`)

- C allows defining a generic pointer by

  ```
  void * p;
  ```

  `p` is a simple address of a memory location, however no type of the pointed variable is specified

- It is possible to have

  ```
  int v=4;
  void * p;

  p = &v;
  ```

  however, it is not possible to dereference it by `*p`. The compiler doesn't know how to interpret the byte at the memory location pointed by `p`.

Pointer arithmetics

- If `p` is a pointer to `<type>`, `(p+i)` is a pointer to `p[i]` of the array `p` of elements of type `<type>`
- The address pointed by `p+i`, then is `p+i*dim`, with `dim=sizeof(*p)`
- Example: assuming that the following variables are declared

  8100 ·4

  `int v[10] = {1, 9, 1000}, *q = v + 3;` =810C (exa)

  among the following expressions, which one is correct?

  For the correct ones, what is the action taken?

8104

`q = v+1;`

`v = q+1;` No

`q++;` 8110

`*q = *(v+1);` $9 \xrightarrow{\text{is}} v[3]$

`*q = *v+1;` 2 → v[3]

`q[4] = *(v+2);` 1000 → v[7]

`v[1] = (int)*((char *)q-3);` v[1]=3

`q[-1] = *(((int *)&q)-9);` v[-1] = v[-6]

`v[-1] = *(--q);` v[-1] = v[4]

address	content	variable
exa ...	...	...
008100	1	v
008104	9	
008108	03E8=1000_{10}	
00810C	0	
...	...	
008124	0	
008128	00810C	q
...	...	

Outline

1. C: pointers to memory
    - `scanf`, copying memory

`scanf`: a `printf`-like method to read the input

- `fgets(...)+strtol(...)` require to invoke two functions and a preallocated string buffer
- `scanf` allows to read from `stdin` a string and stores the converted input into the pointed variable
- Standard example of usage

```
int n;

scanf("%i", &n);
```

- `'i'`: reads an integer(hex: if it starts with 0x, octal: it starts with 0, decimal: otherwise)
- Input format is similar to the `printf`
- The input is read until a “white-space”: space, tab, newline
- do not use `scanf` with “%s” to read a string: you may get a segmentation fault (by writing over more than the allocated memory). `fgets` should be used to read strings
- `man scanf` for more format conversions and specifications

Copying/setting memory blocks

- To use the following function, you must add the following line on top of your program

```
#include <string.h>
```

- To copy n bytes from the memory pointed by src to the memory pointed by dst, we can use

```
void *memcpy(void *dest, const void *src, size_t n);
```

  - we must have access to both *src and *dest
  - troubles if two memory areas overlap (check bcopy(...) or memmove(...) in case of overlap)

- To fill the first n bytes pointed by p with the character c, use

```
void *memset(void *p, int c, size_t n);
```

  - the memory area pointed by p must be allocated
  - bzero(p,n) is the same as memset(p, 0, n)