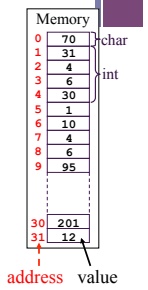
Pointers and Arrays

Today's Goals

- Pointers
  - Declaration
  - Assignment
  - Indirection/de-referencing
- Arrays
  - 1D arrays
  - pointer arithmetic
  - 2D arrays

Variable and Address

- Variable = Storage in computer memory
  - Contains some value
  - Must reside at a specific location called *address*
  - Basic unit – byte
  - Imagine memory as a one-dimensional array with addresses as byte indices
  - A variable consists of one or more bytes, depending on its type (size)



Pointer – Reference

- A pointer (pointer variable) is a variable that stores an address (like Java reference)
  - value – address of some memory
  - type – size of that memory
- Recall in Java, when one declares variables of a *class* type, these are automatically references.
- In C, pointers have special syntax and much greater flexibility.

Memory and Address

- A machine with 16 Megabytes of memory has ? bytes

$$16 \times 2^{20} = 2^4 \times 2^{20} = 16{,}777{,}216$$

- Since each byte has a unique address, there are at least that many addresses
- A pointer stores a memory address, thus the size of a pointer is machine dependent
- With most data models it is the largest integer on the machine, size of `unsigned long`
- Defined in `inttypes.h`
  - `uintptr_t` and `uintmax_t`

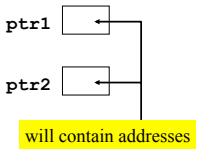
Address Operations in C

- Declaration of pointer variables
  - The *pointer declarator* '`*`'
- Use of pointers
  - The *address of* operator '`&`'
  - The *indirection* operator '`*`' – also known as de-referencing a pointer

Pointer Declaration

7

- Syntax
  - *destinationType* `*` *varName*`;`
- Must be declared with its associated type.
- Examples
  - `int *ptr1;`
    A pointer to an `int` variable
  - `char *ptr2;`
    A pointer to a `char` variable



Pointers are NOT integers

8

- Although memory addresses are essentially very large integers, pointers and integers are not interchangeable.
- Pointers are not of the same type
- A pointer's type depends on what it points to
  - `int *p1; // sizeof(int)`
  - `char *p2; // sizeof(char)`
- C allows free conversion btw different pointer types via casting (dangerous)

Address of Operator

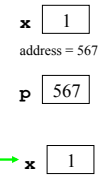
9

- Syntax
  - `&` *expression*
    The expression must have an address. E.g., a constant such as "`1`" does not have an address.
- Example
  - `int x = 1;`
    `f(&x);`
    The address of `x` (i.e. where `x` is stored in memory), say, the memory location 567, (not 1) is passed to `f`.

Pointer Assignment

10

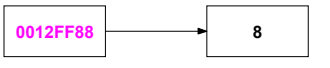
- A pointer `p` *points* to `x` if `x`'s address is stored in `p`
- Example
  - `int x = 1;`
    `int *p;`
    `p = &x;`



Interpreted as:

Pointer Diagram

11



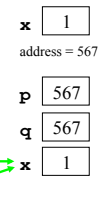
ip i (@0012FF88)

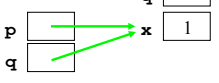
```
int i = 8;
int *ip;

ip = &i;
```

Pointer Assignment

12

- A pointer `p` *points* to `x` if `x`'s address is stored in `p`
- Example
  - `int x = 1;`
    `int *p, *q;`
    `p = &x;`
    `q = p;`
    Interpreted as:



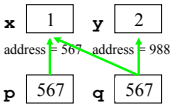


Pointer Assignment

13

- Example

```
int x=1, y=2, *p, *q;
p = &x; q = &y;
q = p;
```



Indirection Operator

14

Note: '*' in a declaration and '*' in an expression are different.

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

- Syntax
  - * *pointerVar*
  - Allows access to value of memory being pointed to
  - Also called *dereferencing*
- Example

```
int x = 1, *p;
p = &x;
cout << *p << endl;
```

*p refers to x; thus prints 1

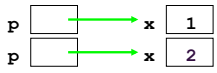
Assignment Using Indirection Operator

15

- Allows access to a variable indirectly through a pointer pointed to it.
- Pointers and integers are not interchangeable
- Example

```
int x = 1, *p;
p = &x;
*p = 2;
cout <<  x << endl;
```

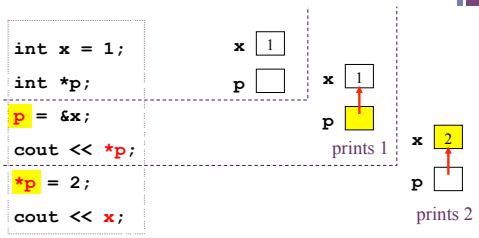
- *p is equivalent to x



Schematically

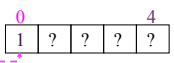
16

```
int x = 1;
int *p;
p = &x;
cout << *p;
*p = 2;
cout << x;
```

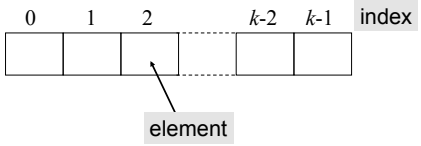


Arrays

- Declaration – `int a[5]; a`
- Assignment – `a[0] = 1;`
- Reference – `y = a[0];`



- Schematic representation

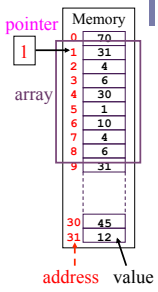


Pointers and Arrays

- Arrays are contiguous allocations of memory of the size:

```
sizeof(elementType)
* numberOfElements
```

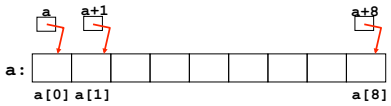
- Given the address of the first byte, using the type (size) of the elements one can calculate addresses to access other elements



18

Name of an Array

- The variable name of an array is also a pointer to its first element.



- `a == &a[0]`
- `a[0] == *a`

19

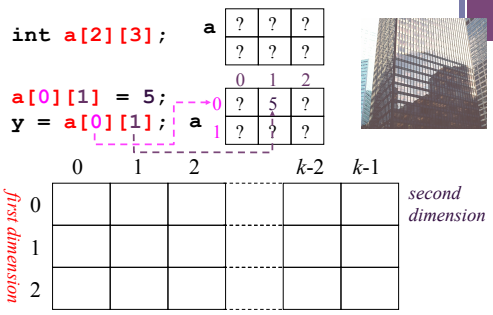
Pointer Arithmetic

- One can add/subtract an integer to/from a pointer
- The pointer advances/retreats by that number of *elements (of the type being pointed to)*
  - `a+i == &a[i]`
  - `a[i] == *(a+i)`
- Subtracting two pointers yields the number of *elements* between them

20

Multi-Dimensional Array

```
int a[2][3];
a[0][1] = 5;
y = a[0][1];
```



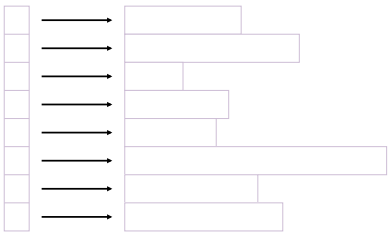
21

Pointer Arrays: Pointer to Pointers

- Pointers can be stored in arrays
- Two-dimensional arrays are just arrays of pointers to arrays.
  - `int a[10][20]; int *b[10];`
  - Declaration for b allows 10 `int` pointers, with no space allocated.
  - Each of them can point to an array of 20 integers
  - `int c[20]; b[0] = c;`
  - What is the type of **`b`**?

22

Ragged Arrays



23

Combining `*` and `++/--`

- `++` and `--` has precedence over `*`
  - `a[i++] = j;`
  - `p=a; *p++ = j; <==> *(p++) = j;`
  - `*p++; value: *p, inc: p`
  - `(*p)++; value: *p, inc: *p`
  - `++(*p); value: (*p)+1, inc: *p`
  - `*++p; value: *(p+1), inc: p`

24

Summary

- Pointer and integers are not exchangeable
- Levels of addressing (i.e. layers of pointers) can be arbitrarily deep
- Failing to pass a pointer where one is expected or vise versa always leads to segmentation faults.
- Understand the relationship between arrays and pointers
- Understand the relationship between two-dimensional arrays and pointer arrays
- Pointer arithmetic is powerful but dangerous!