Pointers

Class 18

Recap

- a variable is allocated exactly enough memory to hold one value of the declared type

```
int value;
```

int 1234 value

```
double price;
```

double 123.4567 price

```
char initial;
```

char 'A' initial

Variables in Memory

- a computer's memory is a list of numbered locations, each of which refers to a byte of 8 bits
- the number of a byte is its address
- a simple variable (e.g., int or double) refers to a portion of memory containing a number of consecutive bytes
- the number of bytes is determined by the type of the variable (e.g., on sand, 4 bytes for unsigned, 8 bytes for double)
- the address of the variable is the address of the first byte where it is located

Address Operator

- when you use a variable in a program, the compiler assumes you want the contents of that variable's location in memory
- in other words, the value stored in that variable

- but sometimes you actually want the address of the variable in memory
- sometimes you also want to know how many bytes of memory a variable occupies
- there is a way to do each of these

- to get a variable's address, we use the address-of operator: &
- to get the number of bytes a variable takes up, we use the sizeof operator (it looks like a function, but it's really an operator)

```
/* illustrate the address-of and sizeof operators */
#include <stdio.h>
int main(void)
{
  int x = 25000;
  double y = 123.4;

  printf("x's address is %p, x = %d, "
         "x is %zu bytes long\n", (void*)&x, x, sizeof x);

  printf("y's address is %p, y = %0.1f, "
         "y is %zu bytes long\n", (void*)&y, y, sizeof y);

  return 0;
}

$ ./program
x's address is 0x7ffffa07db28, x = 25000, x is 4 bytes long
y's address is 0x7ffffa07db20, y = 123.4, y is 8 bytes long
```

Pointer Variables

- a pointer variable aka pointer is a variable that holds a memory address

- just as the purpose of an `int` is to hold an integer
- and a `double` is to hold a double
- the purpose of a pointer is to hold an address
- this allows you to indirectly reference a stored value though the use of a variable that "points to" another location

Reference Variables

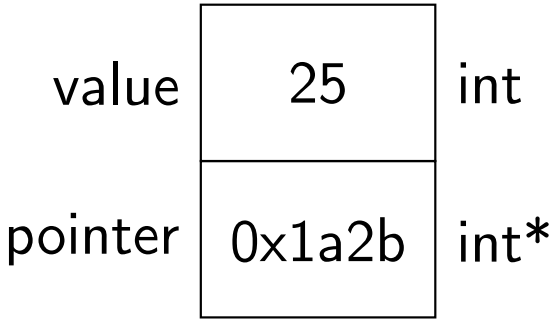
- in C++ and Java, you used variables that refer to other locations
- a reference parameter is an alias for the "real" variable that is located in the calling scope
- in Java, every object variable is a reference variable that refers to the place in memory where the object is stored

- pointers are very similar to references, but operate at a lower level
- almost all the details of references are done for you by the compiler
- pointers require you to do all the details yourself

Declaring a Pointer

```
int value = 25;
int* pointer = &value;

printf("value: %d\n", value);
printf("value's address: %p\n",
       (void*)pointer);
```



note that K&R almost always leave the data type out of diagrams, but we always include them

A Note On Style

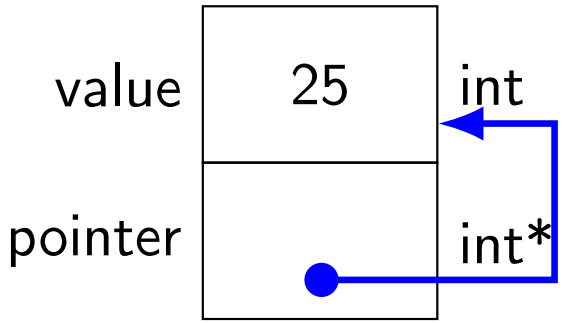
- K&R declare a pointer variable like this:
  `int *ip;`
  with the asterisk attached to the variable name
- they argue this makes sense because the data type of the expression "`*ip`" is an int

- I instead use this declaration:
  `int* ip;`
  with the asterisk attached to the data type
- because the data type of the variable "`ip`" is "integer pointer"
- I believe that the variable `ip` is a more fundamental construct than the expression `*ip`

- you can do either, but be consistent, and never do this:
  `int * ip;`

Depicting a Pointer

- usually instead of writing the actual address value
- we show the address symbolically with an arrow
- this shows the pointer variable pointing to a memory location



Using a Pointer Variable

- once a pointer variable has a valid value, it can be used
- the value in the pointer variable itself is an address, usually not directly useful
- to get at the value the pointer is pointing to, we must dereference it using the dereference operator *

```
1 int value = 5;
2 int* pointer = &value;
3
4 value++;
5 *pointer += 5;
6 printf("value is %d\n", value);
7 printf("pointer points to %d\n",  *pointer);
```

make sure you can draw a picture of memory

Variable Size

- all the variables we have declared so far are exactly large enough for one value of the declared type

```
int value;
```

int 1234 value

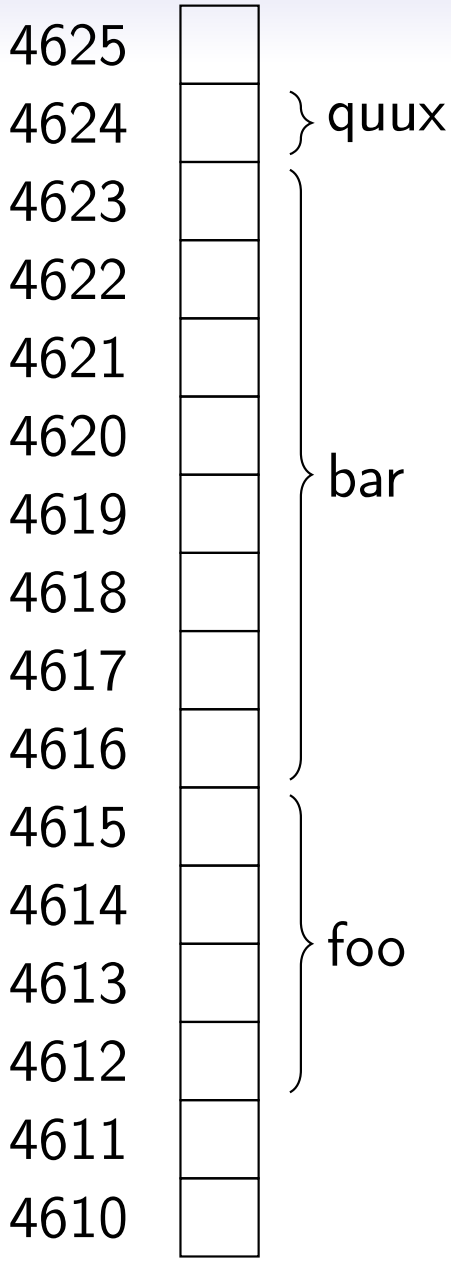
```
double price;
```

double 123.4567 price

```
char initial;
```

char 'A' initial

```
int main(void)
{
 unsigned foo; /* address 4612 */
 double bar; /* address 4616 */
 char quux; /* address 4624 */
 ...
}
```



Array Variable

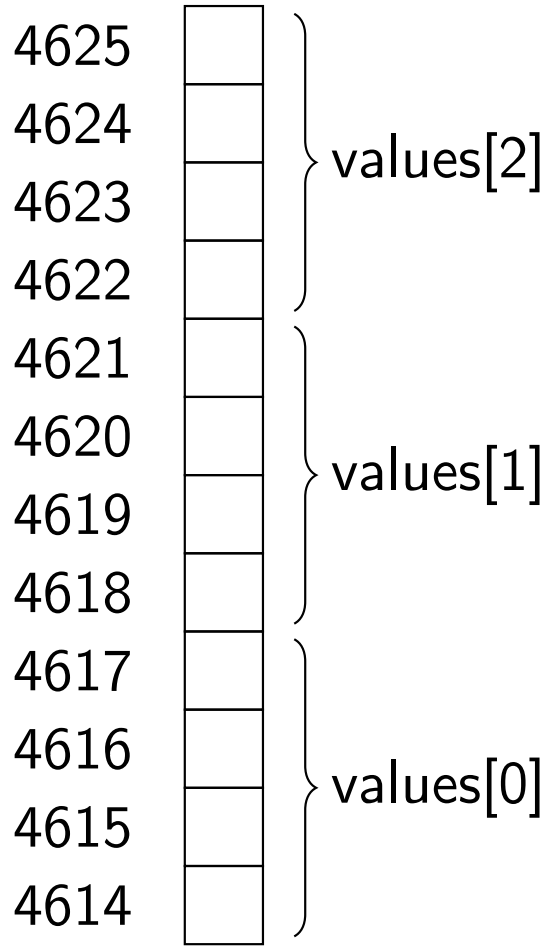
- an array acts like a variable that can store many values
  - all of the same type
  - contiguously, one after the other, in memory

```
#define NUMBER_OF_VALUES 3
int values[NUMBER_OF_VALUES];
```

- allocates enough memory to hold three integers

```
#define NUMBER_OF_VALUES 3
int values[NUMBER_OF_VALUES];
```

- the address of `values[2]` is 4622
- the address of values is 4614, the same as `values[0]`
- both values and values[0] refer to the same location
- but values[0] means the contents of one element of the array, while values is a synonym for 4614



The Value of the Array Variable Itself

```
int main(void)
{
  int values[] = {10, 20, 30};

  printf("%d\n", values[0]);
  printf("%p\n", (void*)values);
}
```

- when run on borax, the literal output is:

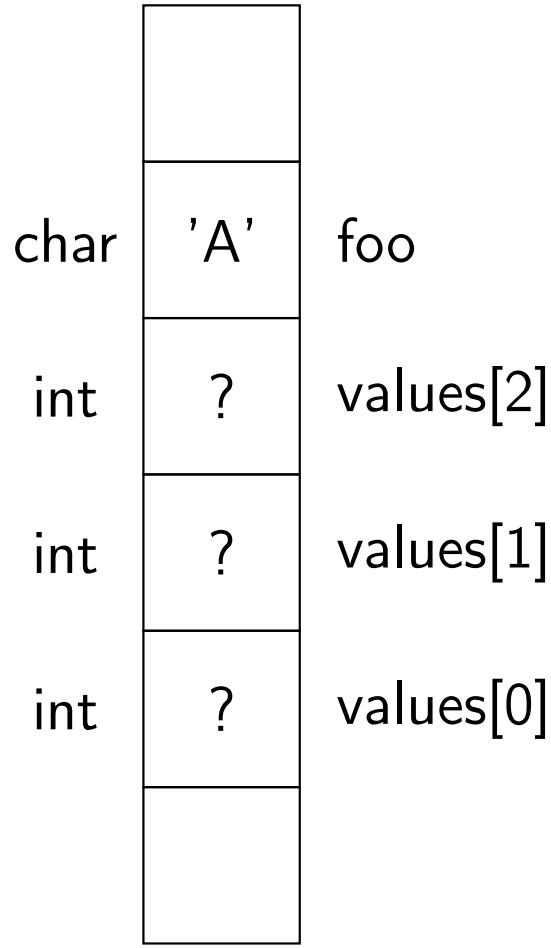
```
10
0x7ffd947d5d40
```

- the latter is the actual physical address in hexadecimal of the location in memory where the first element of values is stored

```
#define NUMBER_OF_VALUES 3
int values[NUMBER_OF_VALUES];
char foo = 'A';
```

- normally, however, we do not show the individual bytes of a variable
- or even the exact address values

- there is no `values[3]` but if there were, it would be above `values[2]`, where foo is



Arrays

```
double temperatures[100]; /* can hold 100 doubles */
unsigned counts[500];     /* can hold 500 unsigned ints */
```

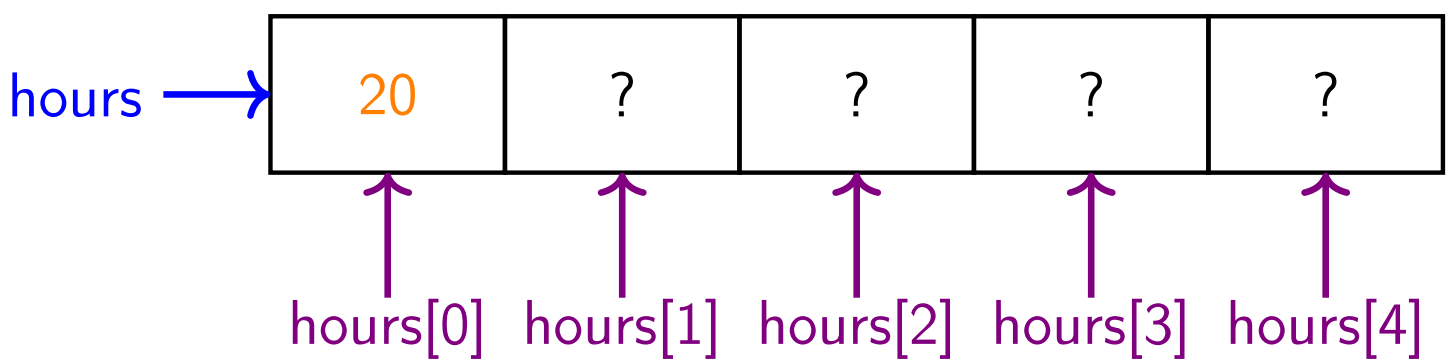
- the amount of RAM used by an array is exactly the number of bytes for one element times the number of elements
- `double temperatures[1000];`
  on sand would consume

$$8 \text{ bytes per double} \times 1000 \text{ doubles} = 8000 \text{ bytes}$$

Array Elements

- the entire array has one name
- individual elements can be accessed using subscripts
- every element in every array is numbered
- the numbers always start at 0 and go up, so they are always unsigned integers of type `size_t`
- a subscript is an unsigned integer expression in square brackets following the name

```
unsigned hours[5];
hours[0] = 20;
```



Arrays

- there are two different types associated with an array
  1. the index type: since indices start at 0 and go up, the index type is always a size_t type
  2. the element type: this can be any type, e.g., int, double, unsigned
- do not confuse the two

Arrays

- there are two different types associated with an array
  1. the index type: since indices start at 0 and go up, the index type is always a size_t type
  2. the element type: this can be any type, e.g., int, double, unsigned
- do not confuse the two

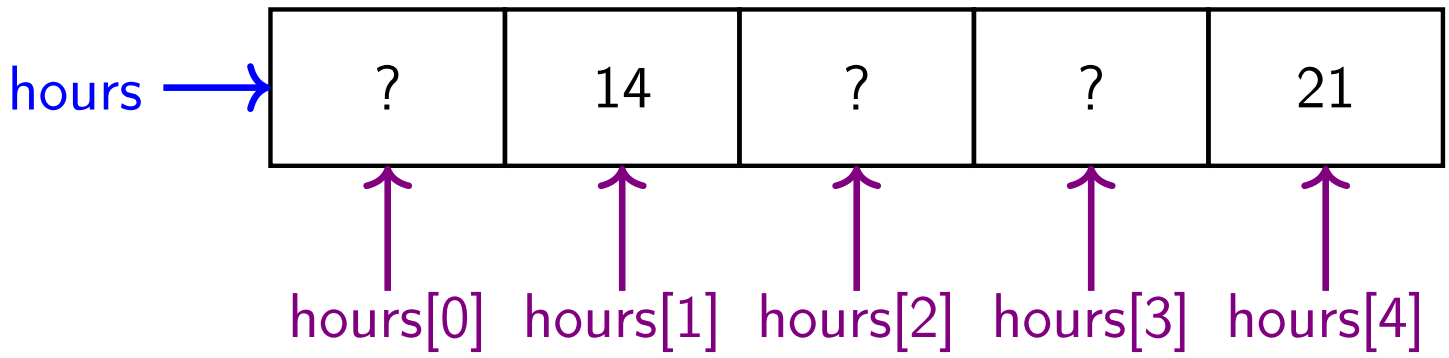
- you cannot use a variable to declare an array's size

```
unsigned score[number_of_scores];
```

- an array's size must be specified by a literal or a constant (or implicit via initialization)
- since a literal will likely be a magic number, use a constant to declare an array's size

Initializing Individual Elements

```
#define ARRAY_SIZE 5
unsigned hours[ARRAY_SIZE];
```



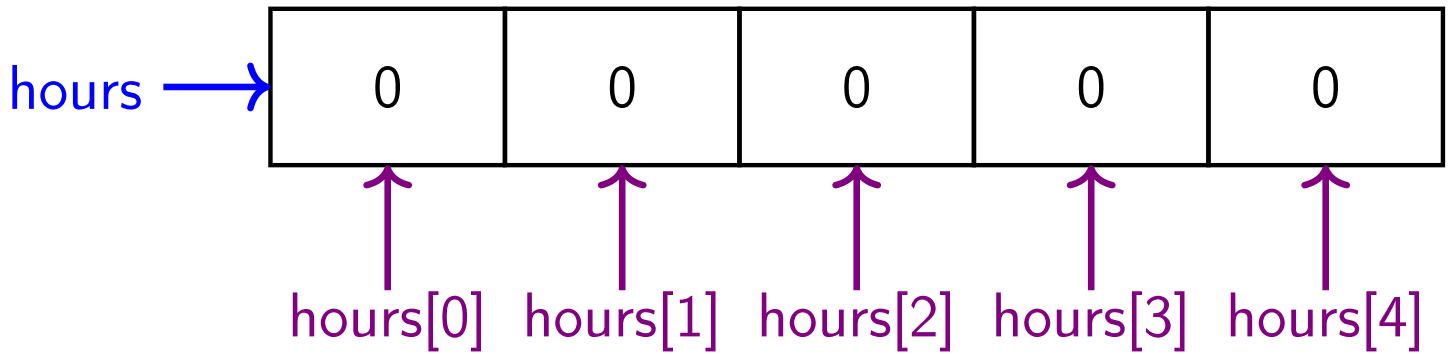
- just like every other variable, array elements are not initialized until the program specifically gives them a value
- they can be given values individually one-by-one:

```
hours[1] = 14;
hours[4] = 21;
```

Initializing Individual Elements

- or in a loop:

```
for (index = 0; index < ARRAY_SIZE; index++)
{
  hours[index] = 0;
}
```



- note: it is rare to have a program with an array that doesn't use loops — for loops and arrays go together like bears and honey

Initialize the Array

- the phrase initialize a variable normally means at the time of declaration
- an array can be initialized at declaration

```
#define NUMBER_OF_MONTHS 12
unsigned days[NUMBER_OF_MONTHS] = {31, 28, 31, 30,
                                   31, 30, 31, 31,
                                   30, 31, 30, 31};
```

- note there is a semicolon after the closing curly brace

Implicit Array Sizing

- if you provide an initialization list, you do not need to specify the size of the array
  ```
  double ratings[] = {1.0, 1.5, 3.3, 2.6, 0.9};
  ```
- the compiler can count the size of the initialization list and know that the full declaration is
  ```
  double ratings[5] = {1.0, 1.5, 3.3, 2.6, 0.9};
  ```

Bounds Checking

- it is illegal to reference an array element that does not exist

```
int foo[10];
foo[10] = 0; /* illegal!  largest index is 9!  */
```

Bounds Checking

- it is illegal to reference an array element that does not exist

```
int foo[10];
foo[10] = 0; /* illegal!  largest index is 9!  */
```

- this will eventually cause you grief
- but sometimes you won't notice it right away

- on sand, the following program will not crash at all with TOO_MANY set to 5
- it will crash (but not immediately) with TOO_MANY set to 20

```
/* Illustrate out-of-bounds */
#include <stdio.h>

int main(void)
{
  #define SIZE  3
  /* #define TOO_MANY  5 */
  #define TOO_MANY 20
  int values[SIZE];
  size_t count;

  for (count = 0; count < TOO_MANY; count++)
  {
    printf("attempting to store in element %zu\n", count);
    values[count] = 100;
  }

  printf("after the first for loop\n");

  for (count = 0; count < TOO_MANY; count++)
  {
    printf("%d  ", values[count]);
  }
  printf("\n");

  printf("after the second for loop\n");
  return 0;
}
```

Pointers and Arrays

- I have emphasized that the array variable name itself, without brackets, is the starting address of the array
- but that's exactly what a pointer is!
- an array name is a pointer
- here, I'll prove it:

```
int numbers[] = {10, 20, 30};
printf("%d\n", *numbers); /* this prints 10! */
```

Pointers and Arrays

- it gets stranger:

```
int numbers[] = {10, 20, 30};
printf("%d\n", *numbers); /* this prints 10! */
printf("%d\n", *(numbers + 1)); /* this prints 20! */
printf("%d\n", *(numbers + 2)); /* this prints 30! */
```

Pointers and Arrays

- it gets stranger:

```
int numbers[] = {10, 20, 30};
printf("%d\n", *numbers); /* this prints 10! */
printf("%d\n", *(numbers + 1)); /* this prints 20! */
printf("%d\n", *(numbers + 2)); /* this prints 30! */
```

- remember, numbers refers to the address of a byte of memory
- but numbers + 1 does not refer to the byte after numbers
- the compiler knows that an int takes up 4 bytes
- thus "numbers + 1" is really "numbers plus enough bytes to get to the next int"
- in other words, "numbers plus sizeof int"

Syntactic Sugar

values[index]

and

*(values + index)

are exactly the same thing

Arrays and Pointers

- array names and pointers are interchangeable
- each printf below prints two identical values

```
double coins1[] = {0.01, 0.05, 0.1, 0.25, 0.5, 1.0};
double* coins2 = coins1;

printf("%f and %f\n", coins1[0], *coins2);
printf("%f and %f\n", coins1[1], *(coins2 + 1));

printf("%f and %f\n", *(coins1 + 2), coins2[2]);
```

Arrays and Pointers

- there is one difference between pointers and array names
- a pointer can be reassigned to point to different things
  but an array name cannot be reassigned

```
int values1[] = {1, 2, 3, 4, 5};
int values2[] = {6, 7, 8, 9, 10};
int* pointer = &values1[2]; /* points to one thing */
pointer = &values2[4];  /* now points to a different thing */
pointer = values1; /* now points to yet another thing */

values1 = pointer; /* illegal! */
                   /* cannot change what values1 points to */
```

- so an array name is a constant pointer

Pointer Arithmetic

- since a pointer stores a numeric value, you can use arithmetic operators on it

```
double coins[] = {0.01, 0.05, 0.1, 0.25, 0.5, 1.0};

double* coin_p = &coins[2]; /* coin_p points to the dime */
coin_p++; /* now points to the quarter */
coin_p -= 2; /* now points to the nickel */
```

- illegal to multiply or divide pointers, can only add and subtract

Comparing Pointers

- pointers may be compared using any of the relops
- what is the value of each of the following expressions?

```
coins1 < &coins1[1]
coins1 < &coins1[4]
coins1 == &coins1[0]
&coins1[2] == coins1 + 2
&coins1[2] != &coins1[3]
```

Comparing Pointers

- pointers may be compared using any of the relops
- what is the value of each of the following expressions?

```
coins1 < &coins1[1]
coins1 < &coins1[4]
coins1 == &coins1[0]
&coins1[2] == coins1 + 2
&coins1[2] != &coins1[3]
```

- this works because array elements are always contiguous in memory

Comparing Pointers

- pointers may be compared using any of the relops
- what is the value of each of the following expressions?

```
coins1 < &coins1[1]
coins1 < &coins1[4]
coins1 == &coins1[0]
&coins1[2] == coins1 + 2
&coins1[2] != &coins1[3]
```

- this works because array elements are always contiguous in memory
- smaller-index elements have smaller addresses than larger-index ones