Arrays

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

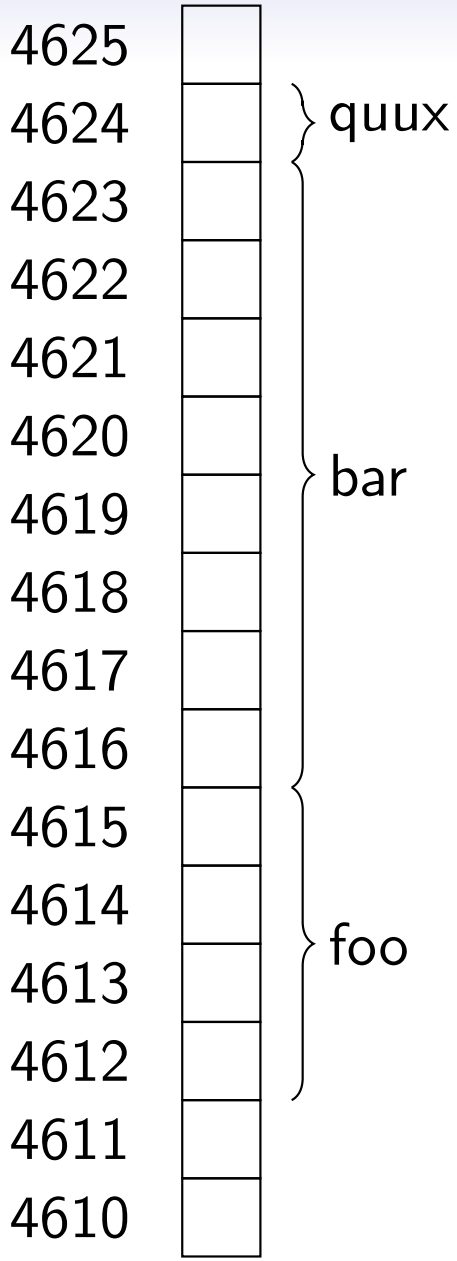
```
char initial;
```

char 'A' initial

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., unsigned or double) refers to a location of memory containing a number of consecutive bytes
- the number of bytes is determined by the type of the variable (e.g., 4 bytes for unsigned, 8 bytes for double)
- the address of the variable is the address of the first byte of memory where it is stored

```
int main()
{
 unsigned foo; // address 4612
 double bar; // address 4616
 bool quux; // address 4624
}
```



Array Variable

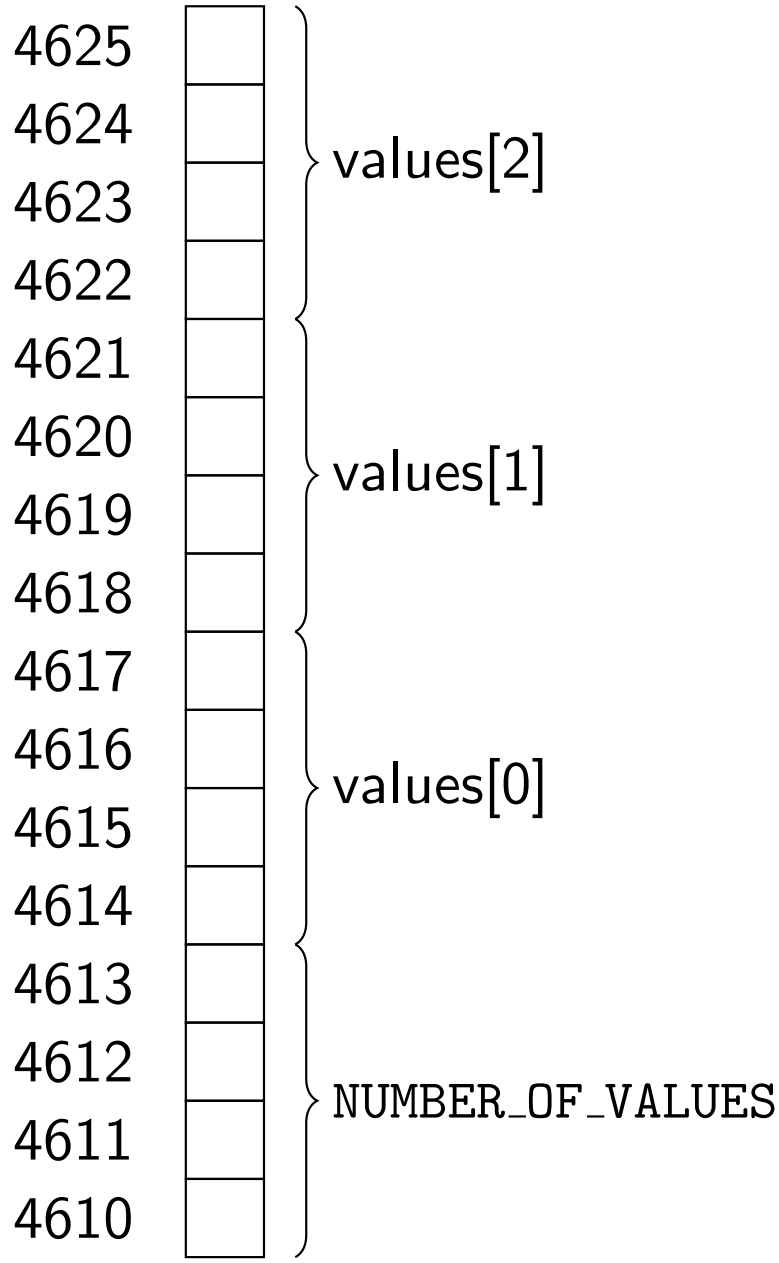
- an array acts like a variable that can store many values
  - all of the same type
  - contiguously, one after the other, in memory

```
const unsigned NUMBER_OF_VALUES = 3;
values[NUMBER_OF_VALUES];
```

- allocates enough memory to hold three integers

```
const unsigned NUMBER_OF_VALUES = 3;
int values[NUMBER_OF_VALUES];
```

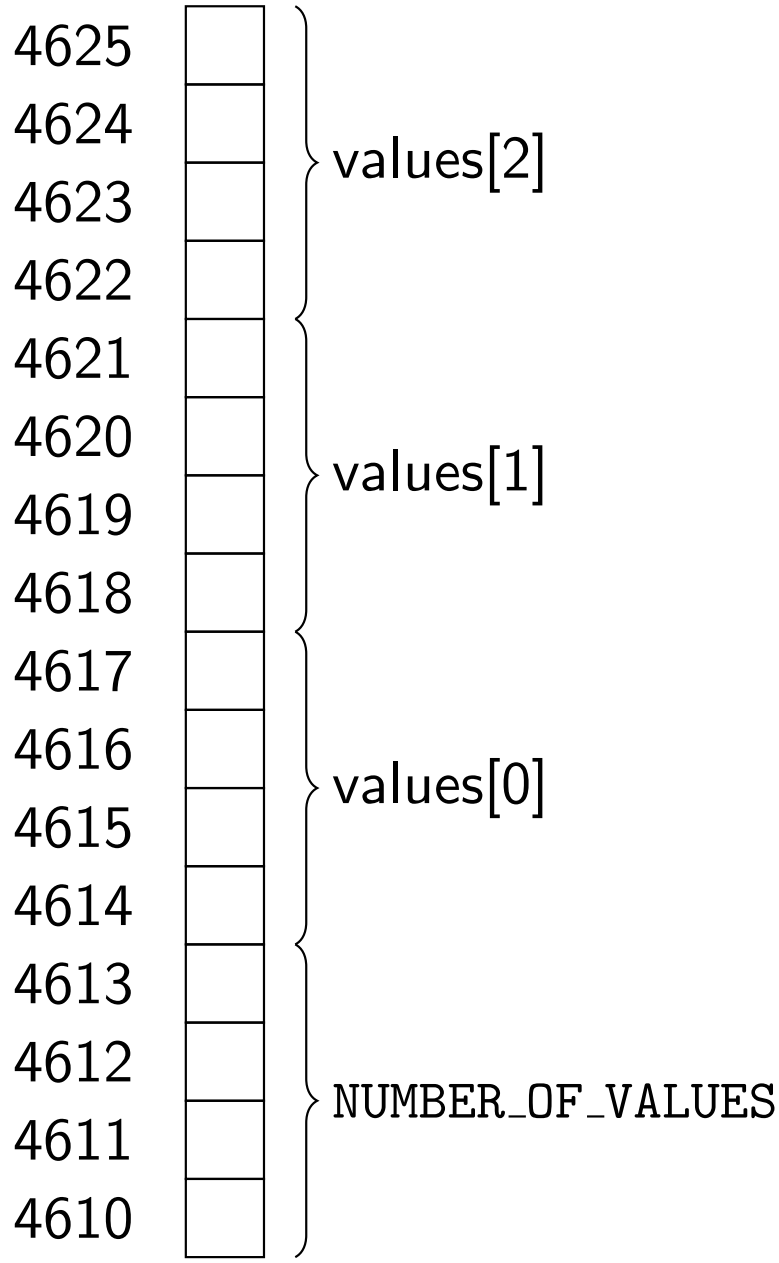
- the address of `NUMBER_OF_VALUES` is 4610
- the address of `values[2]` is 4622



```
const unsigned NUMBER_OF_VALUES = 3;
int values[NUMBER_OF_VALUES];
```

- the address of `NUMBER_OF_VALUES` is 4610
- 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 the address 4614



The Value of the Array Variable Itself

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

  cout << values[0] << endl;
  cout << values << endl;
}
```

- to emphasize, when run on borax, the literal output is:

```
10
0x7ffd947d5d40 // each of you will see a different value
```

- the latter being the actual physical address in hexadecimal of the location in memory of the variable values

Arrays

```
double temperatures[100]; // can hold 100 doubles
string names[50]; // can hold 50 strings
unsigned counts[500]; // can hold 500 unsigned ints
```

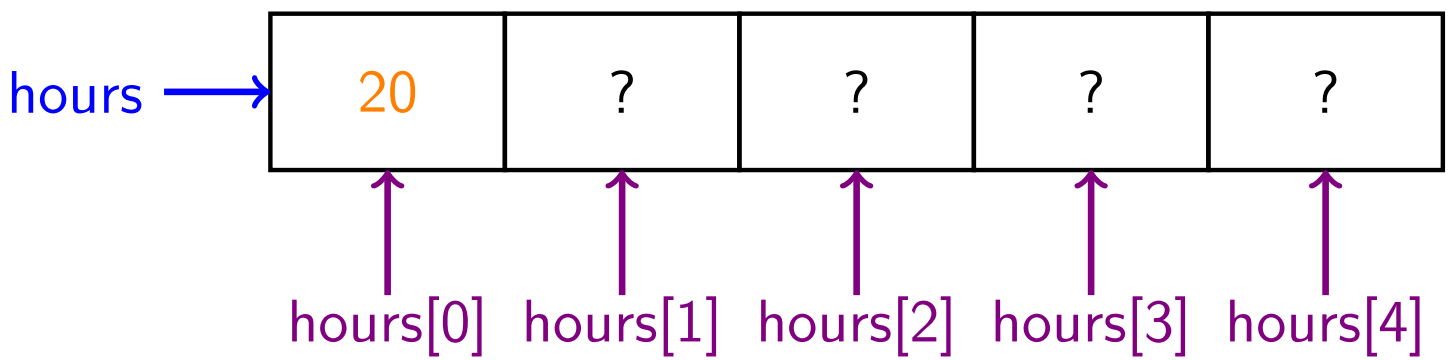
- the amount of RAM used by an array is exactly the number of bytes for one element times the number of elements
- `double temperatures[1000];` on ice would consume

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

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
- 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 an unsigned integer type
  2. the element type: this can be any type, e.g., int, double, string, 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 an unsigned integer type
  2. the element type: this can be any type, e.g., int, double, string, unsigned
- do not confuse the two

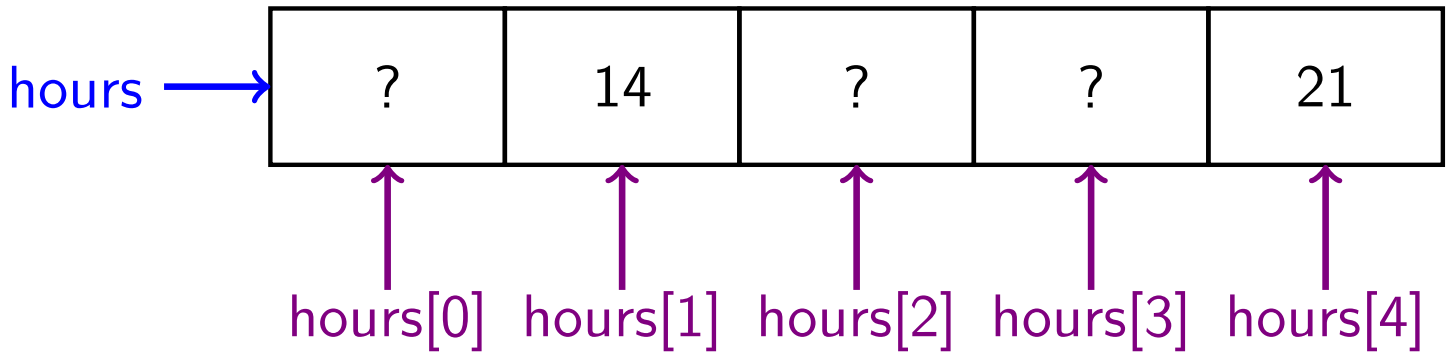
- you cannot use a variable to declare an array's size

```
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

```
const unsigned ARRAY_SIZE = 5;
unsigned hours[ARRAY_SIZE];
```



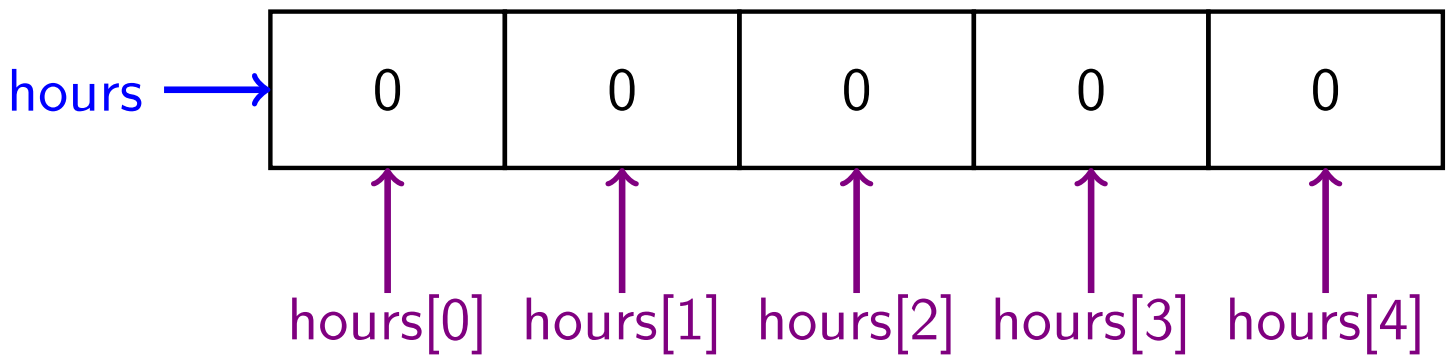
- just like every other variable, array elements are not initialized until the program specifically gives them a value
- they can be given values individually one-by-one:

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

Initializing Individual Elements

- or in a loop:

```
for (unsigned 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

Initialize the Array

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

```
const unsigned 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

see program_7_5.cpp

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!
```

- We have to use caution when using Arrays

- for example, program 7 9.cpp will be in illegal state when TOO MANY set to any value greater than 3; However, the program might still work!