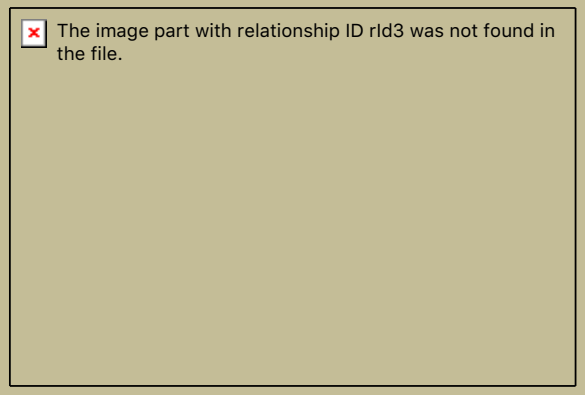
COMPUTER SCIENCE 1: STARTING COMPUTING
CSCI 1300



Ioana Fleming / Vipra Gupta
Spring 2018
Lecture 14

University of Colorado
Boulder

Agenda

- Today
  - Arrays

Announcements

- Rec 5 due on 2/17
- Hmwk 4 (1st Project) due 2/18
- Practicum 1: February 21st, 2018
  - In lecture. 50 minutes.

  or

  - At 6pm LIMITED SPOTS! in ECCR 265

  or

  - At 5pm for Extended-time Accommodations in ECCE 141

Here is what will be on the Practicum

- Multiple Choice Questions (4 or 5 questions)
- Write 3 functions to solve 3 tasks
- Tasks will require you apply
  - Declare functions (return value, parameters)
  - Declare and assign values to variables
  - Use boolean expression for conditionals
  - IF statements
  - IF-ELSE statements
  - Nested IF statements (IF-ELSE IF - …)
  - Iteration via a WHILE statement

Practicum - logistics

- Bring your student ID (knowing your number is not enough)
- 50 minutes
- No cheat sheet
- No communication: no phones, smart watches, messaging apps, no windows open except Moodle and Cloud9
- You have access to all previous created solutions in Cloud9

Practice, practice, practice!

- Review all previous Moodle programming questions from previous recitation and homework assignments
- Review examples we did in class
- Time is short; prepare accordingly.
- Two Practice Practicum Quizzes:
  - one with programming questions
  - one with multiple-choice questions

Tips for Timed Exam

- Read the Questions
  - read them not once, but **TWICE** before starting the code
  - follow all the instructions explicitly (especially for names and order of parameters)
- Create or Modify a Code
  - know your C++ syntax
- Create and Use an IF, IF ELSE
  - know your C++ syntax
  - know how to create a condition
- Create and Use a WHILE
  - know your C++ syntax
  - know how to iterate through a string's characters
- Passing parameters
  - know your C++ syntax

Arrays

- Become familiar with using arrays to collect values
- Learn about common algorithms for processing arrays
- Write functions that receive and return arrays
- Be able to use two-dimensional arrays

Using Arrays

Think of a sequence of data:

32 54 67.5 29 35 80 115 44.5 100 65

(all of the same type, of course)
(storable as **`double`**s)

Using Arrays

32 54 67.5 29 35 80 115 44.5 100 65

Which is the largest in this set?

(You must look at every single value to decide.)

Using Arrays

32 54 67.5 29 35 80 115 44.5 100 65

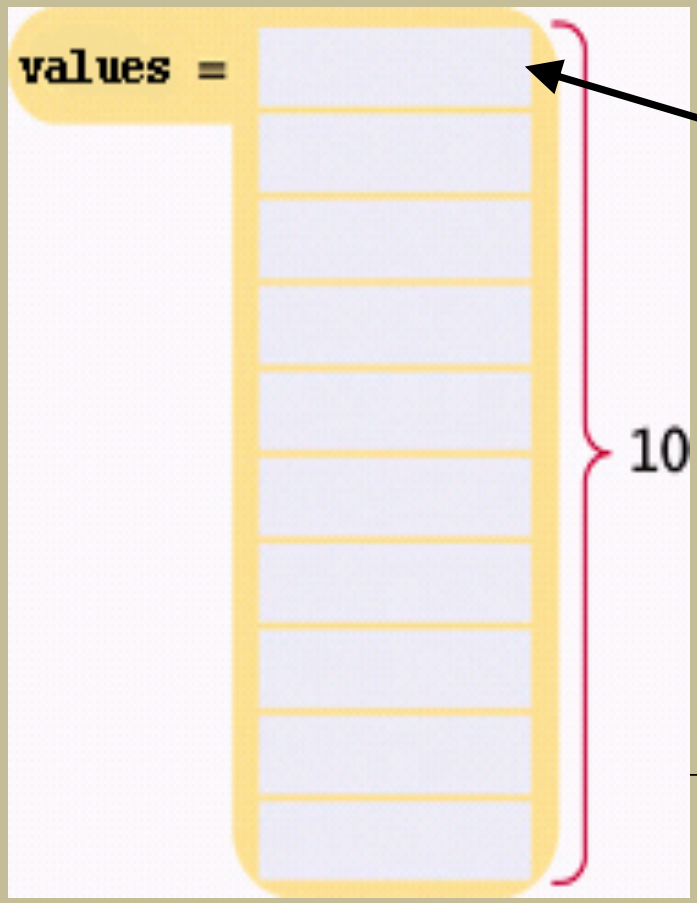
So you would create a variable for each, of course!

```
int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;
```

Then what ???

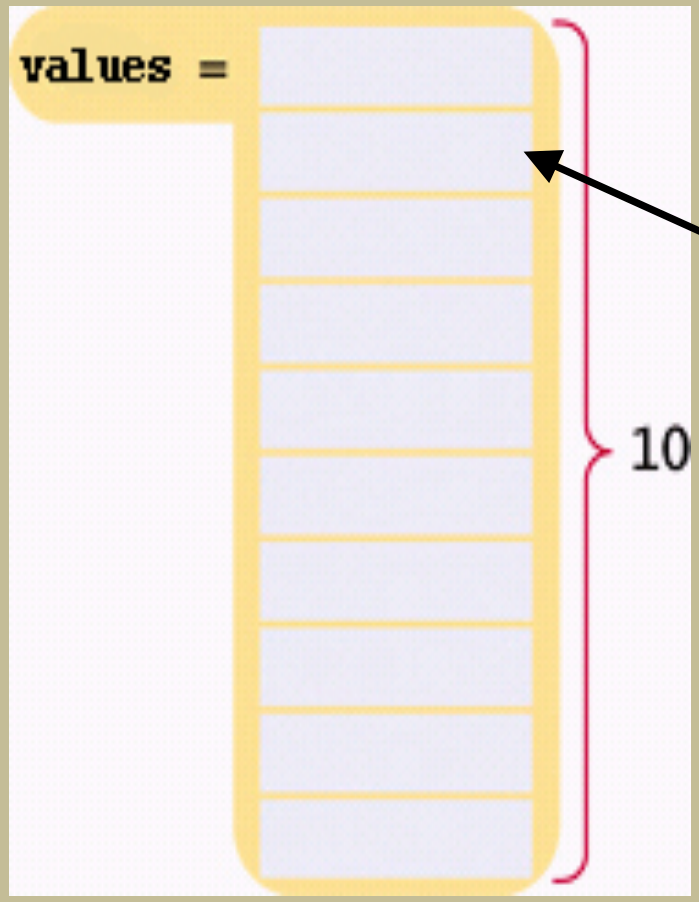
Using Arrays

You can easily visit each element in an array,
checking and updating a variable holding the current maximum.



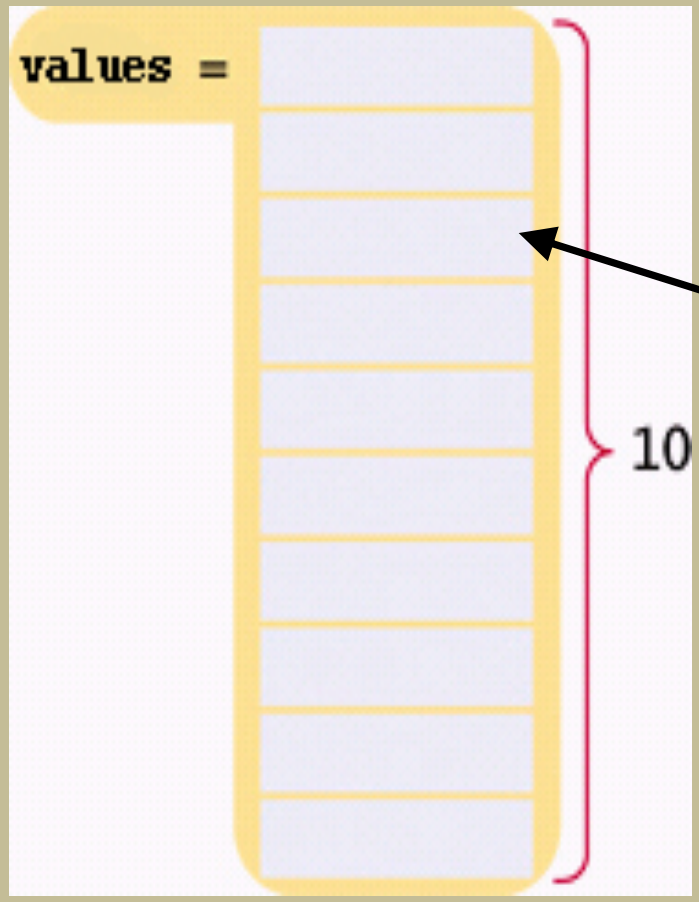
Hm. Is this the max, so far?

Using Arrays

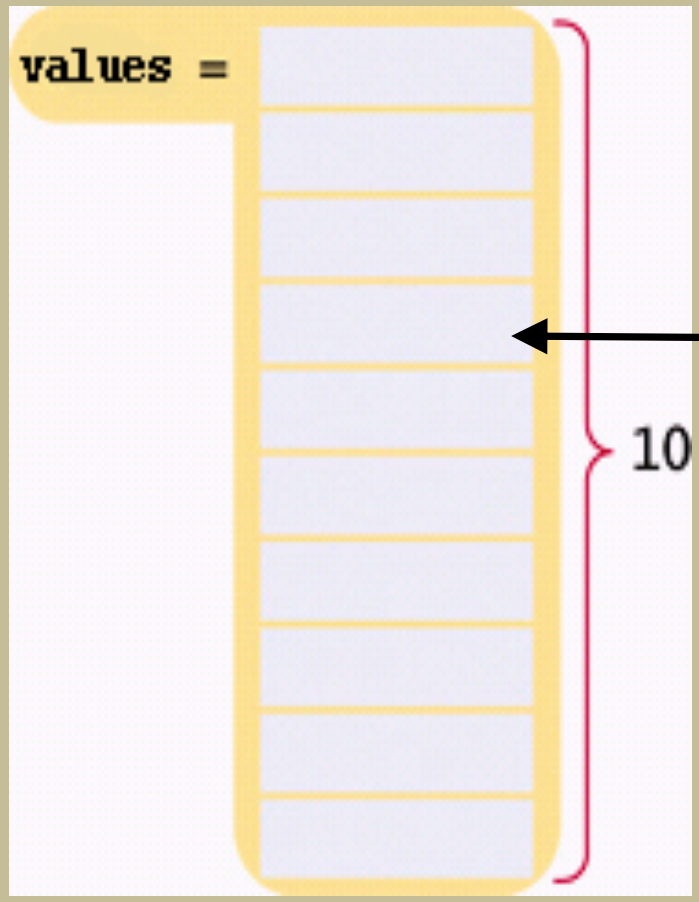


Maybe this one?

Using Arrays

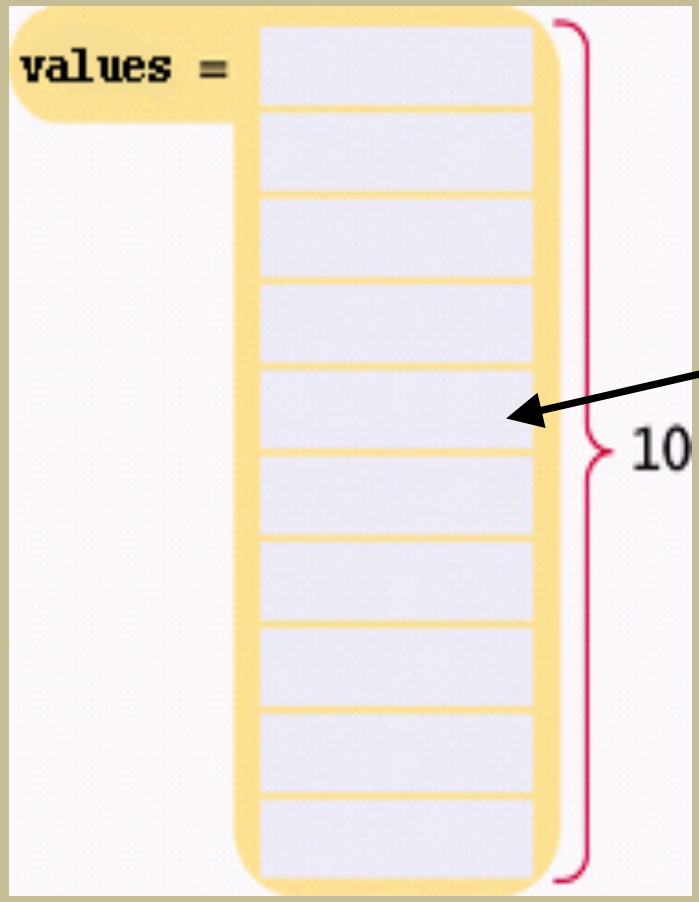


Using Arrays



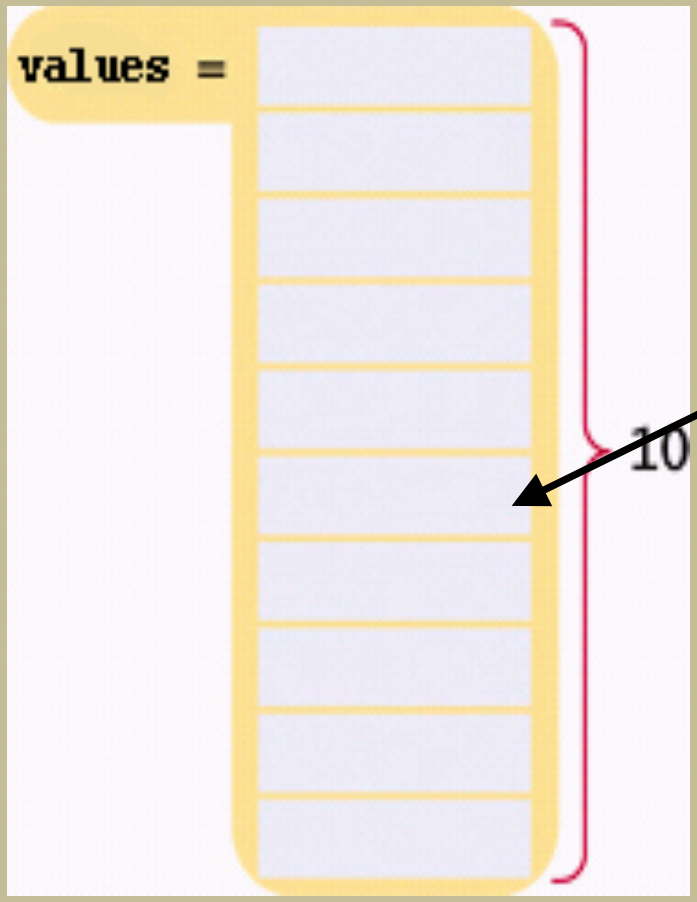
Or this one?

Using Arrays



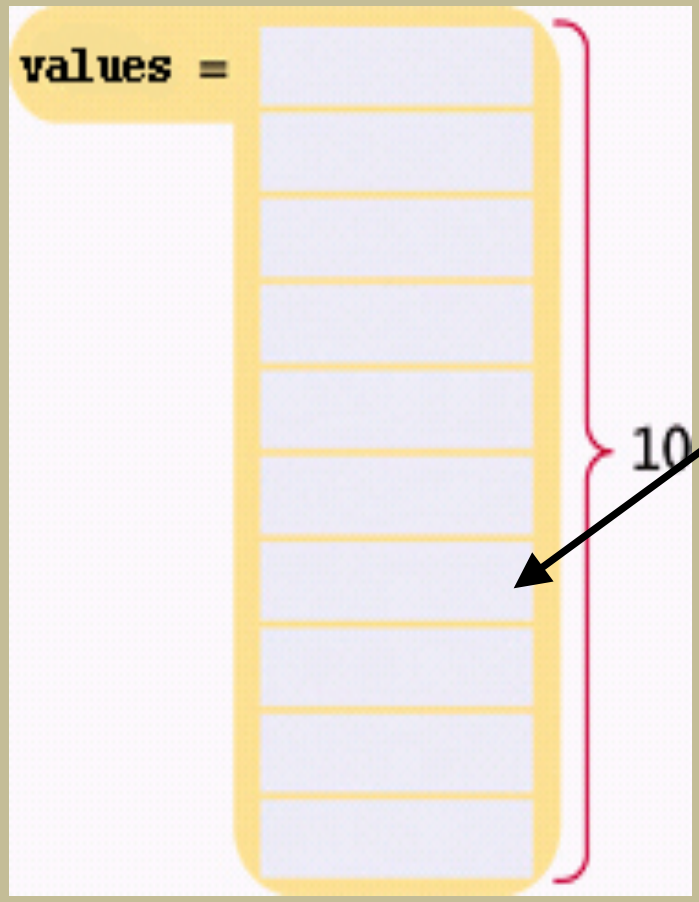
How about here?

Using Arrays



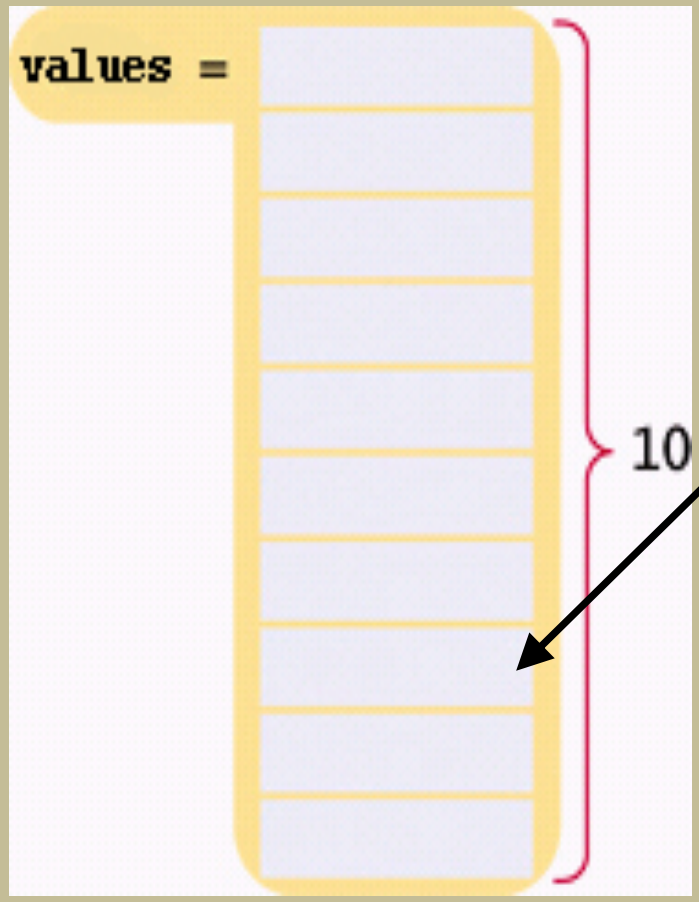
Gotta check here too!

Using Arrays



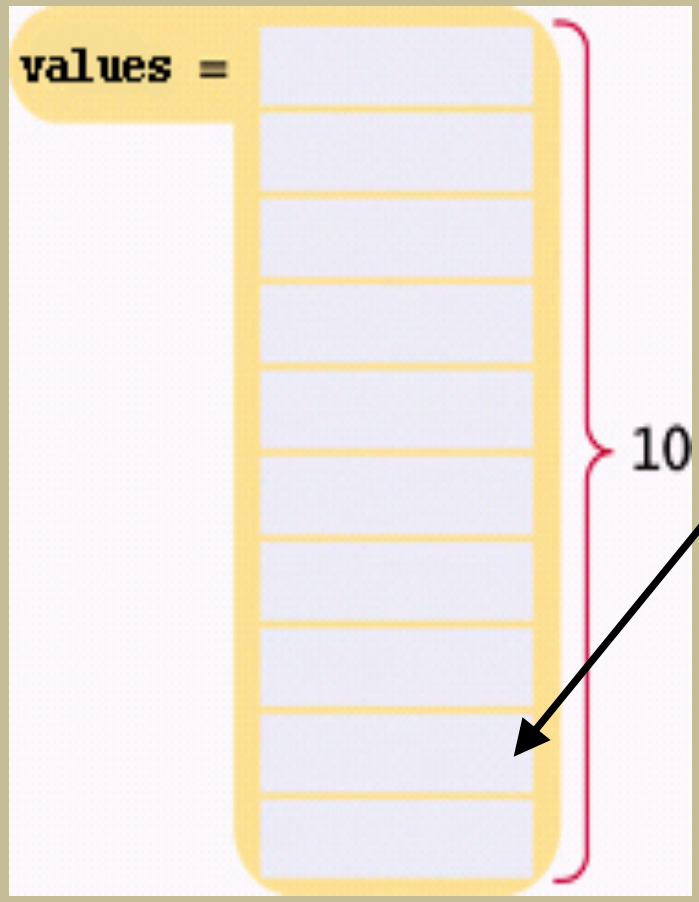
Again, maybe this one?

Using Arrays



Or this one?

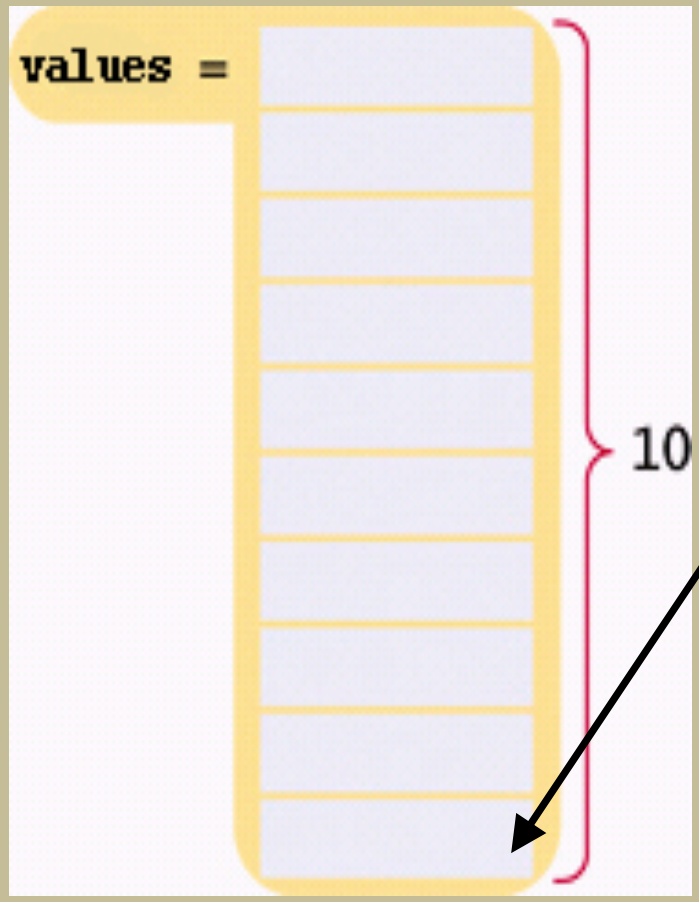
Using Arrays



Or this one?

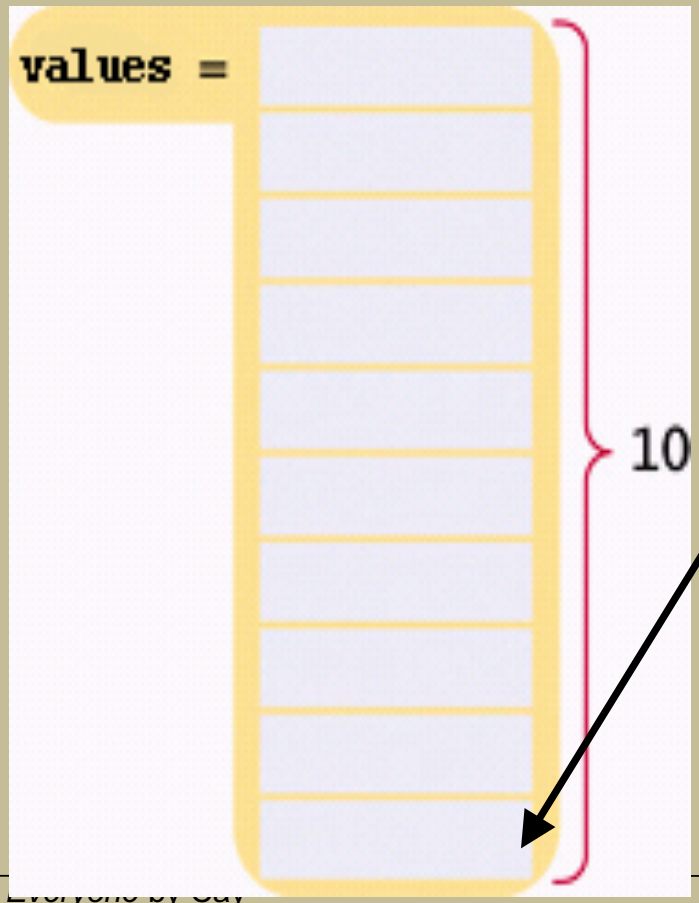
Will this never end?

Using Arrays



Or the last one? Finally!

Using Arrays and Vectors



Or the last one? ***Finally!***

Using Arrays and Vectors

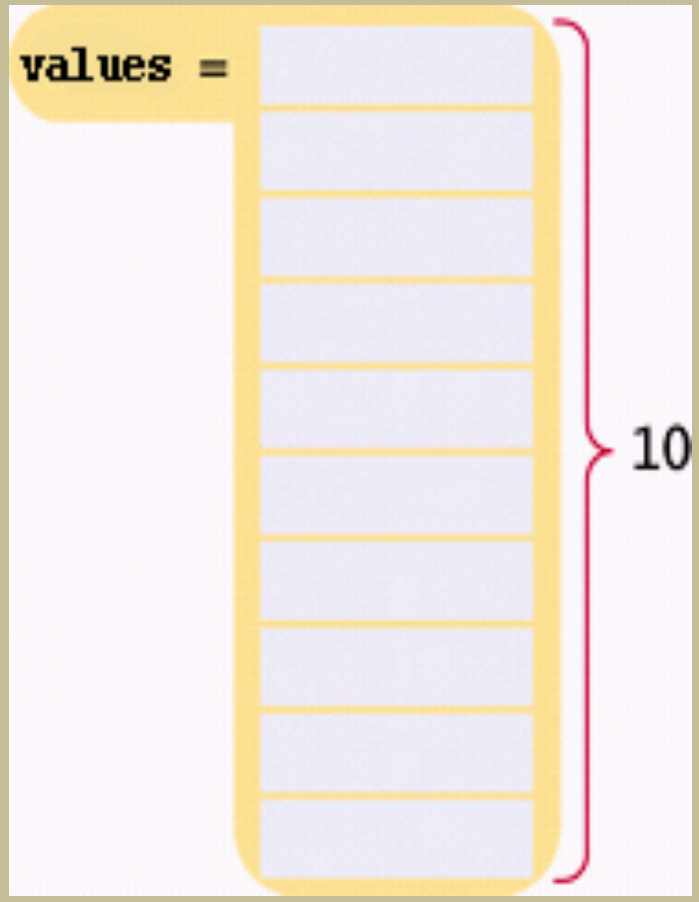
That would have been impossible with ten separate variables!

```
int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;
```

And what if there needed to be more double values in the set?

ARGH!

Defining Arrays



An “array of double”

Ten elements of **`double`** type

can be stored under one name as an array.

```
double values[10];
```

type of each element

quantity of elements – the “size” of the array, must be a constant

Introduction to Arrays

- Array definition:
  - A collection of data of same type
- First "aggregate" data type
  - Means "grouping"
  - int, float, double, char are simple data types
- Used for lists of like items
  - Test scores, temperatures, names, etc.
  - Avoids declaring multiple simple variables
  - Can manipulate "list" as one entity

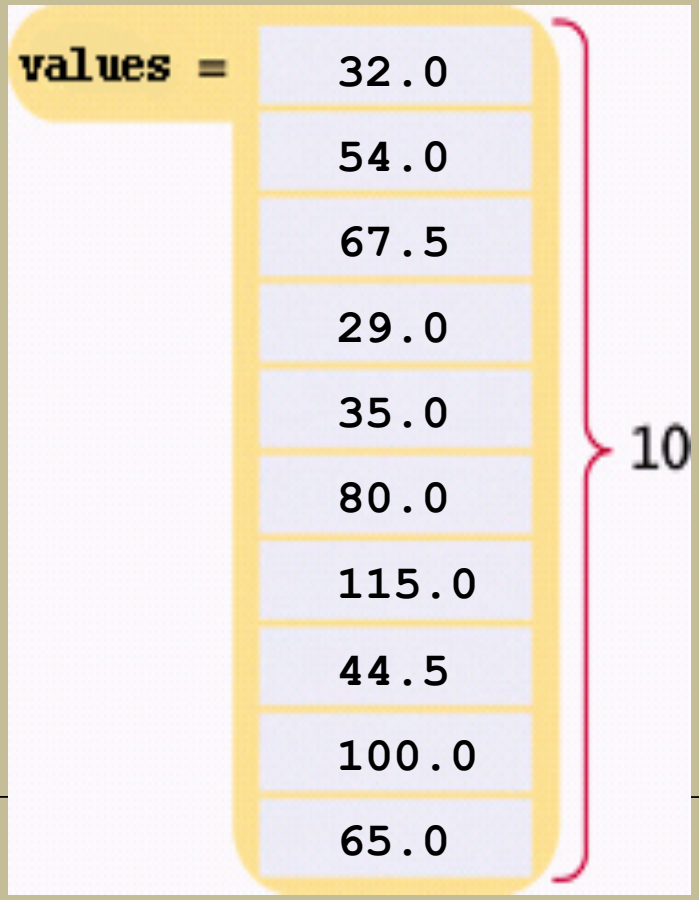
Declaring Arrays

- Declare the array → allocates memory
  `int score[5];`
  - Declares array of 5 integers named "score"
  - Similar to declaring five variables:
    `int score[0], score[1], score[2], score[3], score[4]`
- Individual parts can be called many things:
  - Indexed or subscripted variables
  - "Elements" of the array
  - Value in brackets is called index or subscript
    - Numbered from 0 to (size – 1)

Defining Arrays with Initialization

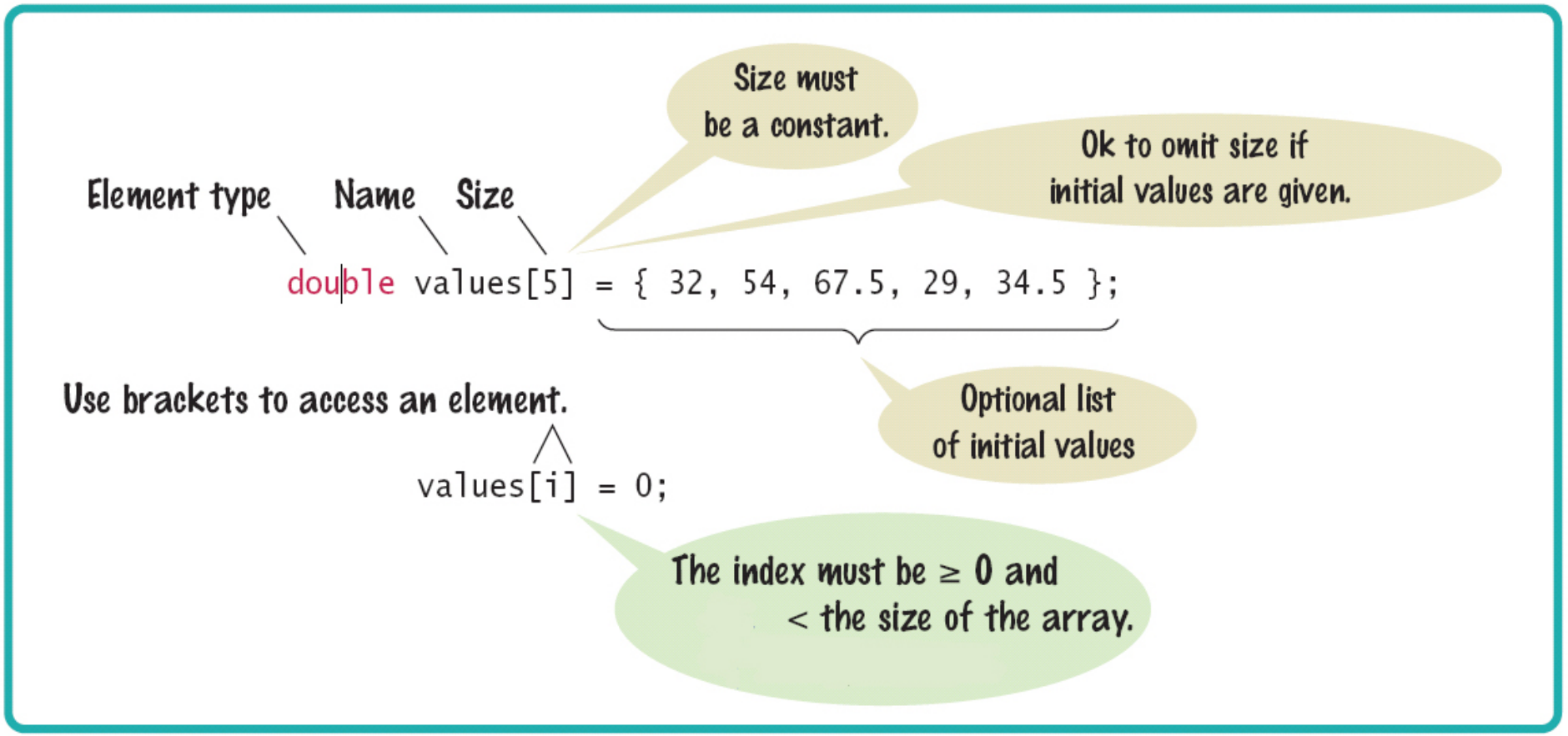
When you define an array, you can specify the initial values:

```
double values[] = { 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 };
```



Array Syntax

Defining an Array



Accessing Arrays

- Access using index/subscript

```
cout << score[3];
```

- Note two uses of brackets:
  - In declaration, specifies SIZE of array
  - Anywhere else, specifies a subscript
- Size, subscript need not be literal

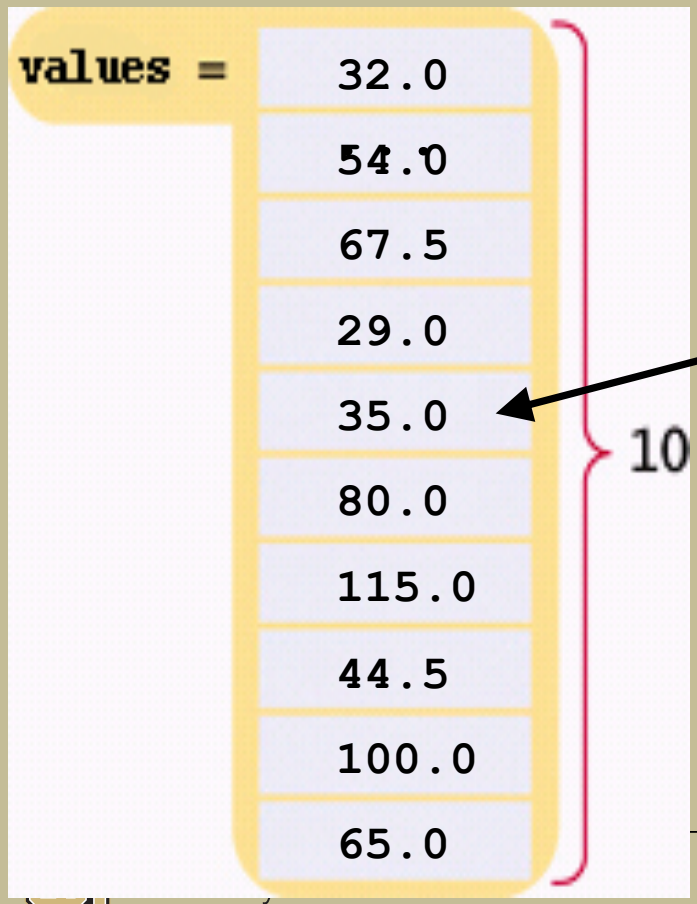
```
int score[MAX_SCORES];
score[n+1] = 99;
```

  - If n is 2, identical to: score[3]

Accessing an Array Element

To access the element at index 4 using this notation: **`values[4]`**

4 is the *index*.



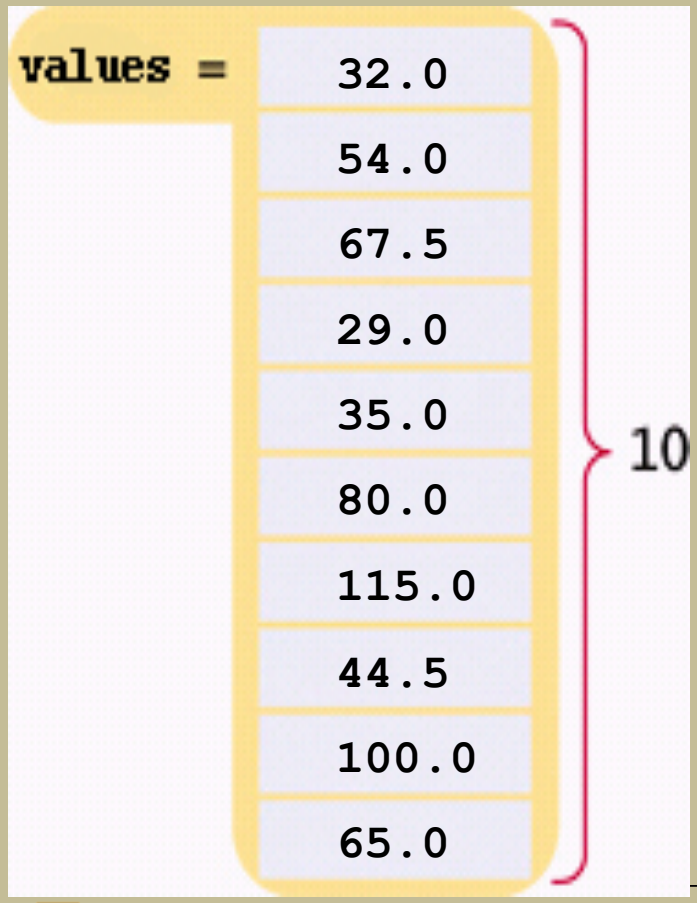

```
double values[10];

cout << values[4] << endl;
```

The output will be `35.0`.

Accessing an Array Element

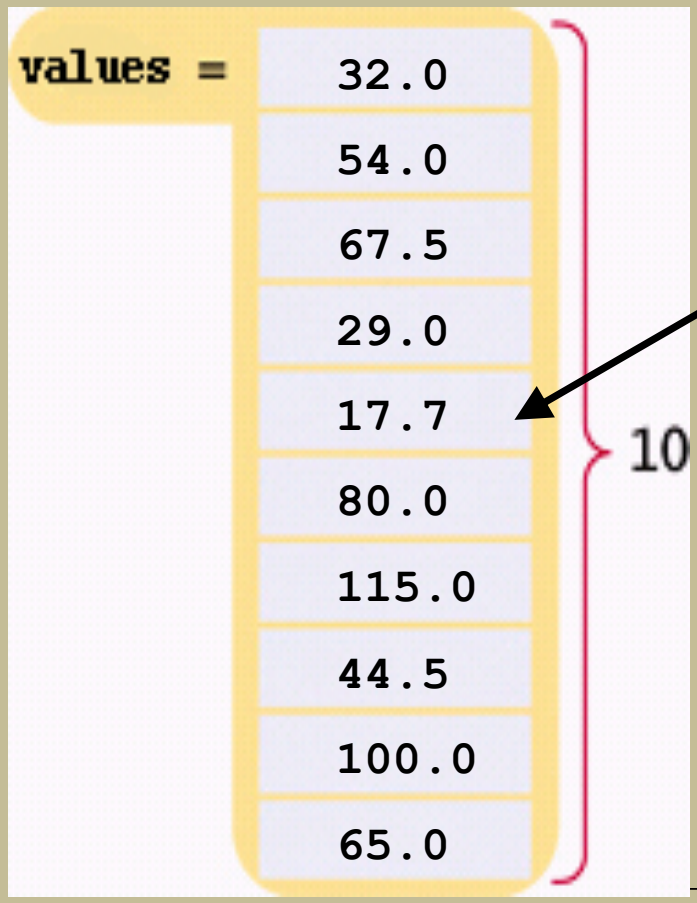
The same notation can be used to change the element.




```
values[4] = 17.7;
```

Accessing an Array Element

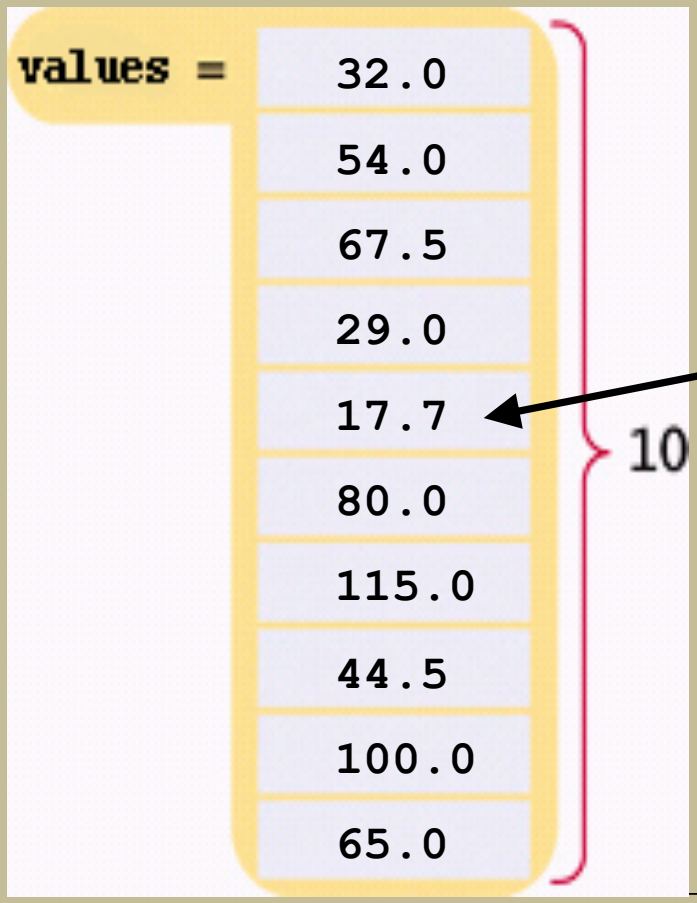
The same notation can be used to change the element.




```
values[4] = 17.7;
```

Accessing an Array Element

The same notation can be used to change the element.




```
values[4] = 17.7;
cout << values[4] << endl;
```

The output will be `17.7`.

Accessing an Array Element

That is, the legal elements for the **`values`** array are:

`values[0]`, the ***first*** element
`values[1]`, the second element
`values[2]`, the third element
`values[3]`, the fourth element
`values[4]`, the fifth element
`...`
`values[9]`, the tenth ***and last legal*** element
recall: **`double values[10];`**

The index must be **`>= 0`** and **`<= 9`**.
0, 1, 2, 3, 4, 5, 6, 7, 8, 9 is 10 numbers.

Array Usage

- Powerful storage mechanism
- Can issue commands like:
  - "Do this to i^{th} indexed variable", where i is computed by program
  - "Display all elements of array score"
  - "Fill elements of array score from user input"
  - "Find highest value in array score"
  - "Find lowest value in array score"
- Disadvantages: size MUST BE KNOWN at declaration

Demo

- Cloud9 – *largest.cpp*

for-loops with Arrays

- Natural counting loop
  - Naturally works well "counting through" elements of an array
- Example:

```
for (idx = 0; idx<5; idx++)
{
    cout << score[idx] << "off by " << max-score[idx] << endl;
}
```

  - Loop control variable (`idx`) counts from 0 – 5

Major Array Pitfall

- Array indexes always start with zero!
- Zero is "first" number to computer scientists
- C++ will "let" you go beyond range
  - Unpredictable results
  - Compiler will not detect these errors!
- Up to programmer to "stay in range"

Major Array Pitfall Example

- Indexes range from 0 to (array_size – 1)
  - Example:

```
double temperature[24];  // 24 is array size
// Declares array of 24 double values called
temperature
```

    - They are indexed as:

```
temperature[0], temperature[1], .. temperature[23]
```

  - Common mistake:

```
temperature[24] = 5;
```

    - Index 24 is "out of range"!
    - No warning, possibly disastrous results

Defined Constant as Array Size

- Always use defined/named constant for array size
- Example:

```
const int NUMBER_OF_STUDENTS = 5;
int score[NUMBER_OF_STUDENTS];
```

- Improves readability
- Improves versatility
- Improves maintainability

Uses of Defined Constant

- Use everywhere size of array is needed
  1. In for-loop for traversal:

```
for (i = 0; i < NUMBER_OF_STUDENTS; i++)
{
     // Manipulate array
}
```

  2. In calculations involving size:

```
lastIndex = (NUMBER_OF_STUDENTS - 1);
```

  3. When passing array to functions (later)

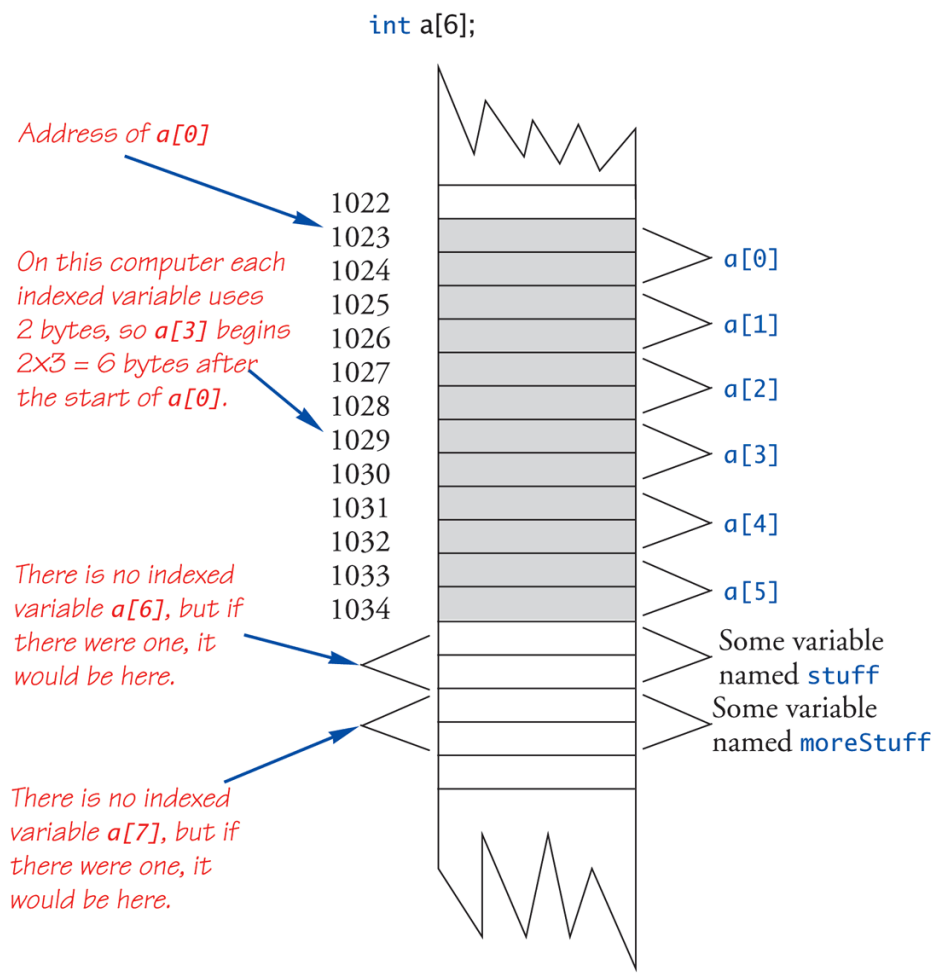
- If size changes → requires only ONE change in program!

Arrays in Memory

- Recall simple variables:
  - Allocated memory in an "address"
- Array declarations allocate memory for entire array
- Sequentially-allocated
  - Means addresses allocated "back-to-back"
  - Allows indexing calculations
    - Simple "addition" from array beginning (index 0)

An Array in Memory

Display 5.2 An Array in Memory



Initializing Arrays

- As simple variables can be initialized at declaration:

```
int price = 0;  // 0 is initial value
```

- Arrays can as well:

```
int children[3] = {2, 12, 1};
```

  - Equivalent to following:

```
int children[3];
children[0] = 2;
children[1] = 12;
children[2] = 1;
```

Auto-Initializing Arrays

- If fewer values than size supplied:
  - Fills from beginning
  - Fills "rest" with zero of array base type
- If array-size is left out
  - Declares array with size required based on number of initialization values
  - Example:

```
int b[] = {5, 12, 11};
```

    - Allocates array b to size 3

Array Syntax

Table 1 Defining Arrays

`int numbers[10];`	An array of ten integers.
`const int SIZE = 10;` `int numbers[SIZE];`	It is a good idea to use a named constant for the size.
 `int size = 10;` `int numbers[size];`	**Caution:** In standard C++, the size must be a constant. This array definition will not work with all compilers.
`int squares[5] = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`int squares[] = { 0, 1, 4, 9, 16 };`	You can omit the array size if you supply initial values. The size is set to the number of initial values.
`int squares[5] = { 0, 1, 4 };`	If you supply fewer initial values than the size, the remaining values are set to 0. This array contains 0, 1, 4, 0, 0.
`string names[3];`	An array of three strings.

Arrays in Functions

- As arguments to functions
  - Indexed variables
    - An individual "element" of an array can be function parameter
  - Entire arrays
    - All array elements can be passed as "one entity”

Indexed Variables as Arguments

- Indexed variable handled same as simple variable of array base type
- Given this function declaration:

```
void myFunction(double par1);
```

- And these declarations:

```
int i;

double n, a[10];
```

- Can make these function calls:

```
myFunction(i);// i is converted to double
myFunction(a[3]);   // a[3] is double
myFunction(n);// n is double
```

Subtlety of Indexing

- Consider:

```
myFunction(a[i]);
```

  - Value of i is determined first
    - It determines which indexed variable is sent

```
myFunction(a[i*5]);
```

  - Perfectly legal, from compiler's view
  - Programmer responsible for staying "in-bounds" of array

Array as argument - details

- What does the computer know about an array?
  - The base type
  - The address of the first indexed variable
  - The number of indexed variables
- What does a function know about an array argument?
  - The base type
  - The address of the first indexed variable

Entire Arrays as Arguments

- Formal parameter can be entire array
  - Argument then passed in function call is array name
  - Called "array parameter"
- Send size of array as well
  - Typically done as second parameter
  - Simple int type formal parameter

Entire Array as Argument Example

- Given previous example:
- In some main() function definition, consider this calls:

```
int score[5], numberOfScores =
5;
fillup(score, numberOfScores);
```

  - 1st argument is entire array
  - 2nd argument is integer value
- Note no brackets in array argument!

Array as Argument: How?

- What's really passed?
- Think of array as 3 "pieces"
  - Address of first indexed variable (arrName[0])
  - Array base type
  - Size of array
- Only 1st piece is passed!
  - Just the beginning address of array

Array Parameters

- May seem strange
  - No brackets in array argument
  - Must send size separately
- One nice property:
  - Can use SAME function to fill any size array!
  - Exemplifies "re-use" properties of functions
  - Example:

```
int score[5], time[10];
fillUp(score, 5);
fillUp(time, 10);
```

The const Parameter Modifier

- Recall: array parameter actually passes address of 1st element
  - Similar to pass-by-reference
- Function can then modify array!
  - Often desirable, sometimes not!
- Protect array contents from modification
  - Use "const" modifier before array parameter
    - Called "constant array parameter"
    - Tells compiler to "not allow" modifications

Example – function definition

```
void addarray(int size,          // IN size of arrays
              const float A[],   // IN input array
              const float B[],   // IN input array
              float C[])         // OUT result array

// Takes two arrays of the same size as input parameters
// and outputs an array whose elements are the sum of the
// corresponding elements in the two input arrays.
{
    int i;
    for (i = 0; i < size; i++)
         C[i] = A[i] + B[i];

  } // End of function addarray
```

Example – function call

The function addarray could be used as follows:

In main():

```
int one[50], two[50], three[50];
            //
            //
addarray(50, one, two, three);

// but also:
addarray(20, one, two, three);

// it will only do the addition on the first 20 elements
of each array
```

Multidimensional Arrays

- Arrays with more than one index
  - char page[30][100];
    - Two indexes: An "array of arrays"
    - Visualize as:
      page[0][0], page[0][1], …, page[0][99]
      page[1][0], page[1][1], …, page[1][99]
      …
      page[29][0], page[29][1], …, page[29][99]
- C++ allows any number of indexes
  - Typically no more than two

Multidimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given
    - Provided as second parameter
  - 2nd dimension size IS given
- Example:

```
void DisplayPage(const char p[][100], int
sizeDimension1)
{
 for (int index1=0; index1<sizeDimension1; index1++)
 {
    for (int index2=0; index2 < 100; index2++)
       cout << p[index1][index2];
    cout << endl;
 }
}
```

Summary 1

- Array is collection of "same type" data
- Indexed variables of array used just like any other simple variables
- for-loop "natural" way to traverse arrays
- Programmer responsible for staying "in bounds" of array
- Array parameter is "new" kind

Summary 2

- Array elements stored sequentially
  - "Contiguous" portion of memory
  - Only address of 1st element is passed to functions
- Partially-filled arrays → more tracking
- Constant array parameters
  - Prevent modification of array contents
- Multidimensional arrays
  - Create "array of arrays"

Programming with Arrays

- Plenty of uses
  - Partially-filled arrays
    - Must be declared some "max size"
  - Sorting
  - Searching

Partially-filled Arrays

- Difficult to know exact array size needed
- Must declare to be largest possible size
  - Must then keep "track" of valid data in array
  - Functions dealing with the array may not need to know the declared size of the array, only how many elements are stored in the array
    - int numberUsed;
    - Tracks current number of elements in array