5. Arrays, Multidimensional Arrays

5.1. Arrays: Reminders

Types (`int`, `float`, `double`, `bool`, `char`, etc.) tell the compiler:

- the size of the variables (e.g., 4, 8, 1 bytes) in memory
- how these bits in memory should be interpreted
- and know the possible operations on them

For example:

if **`height`** and **`width`** are variables of type `int`, then the compiler knows

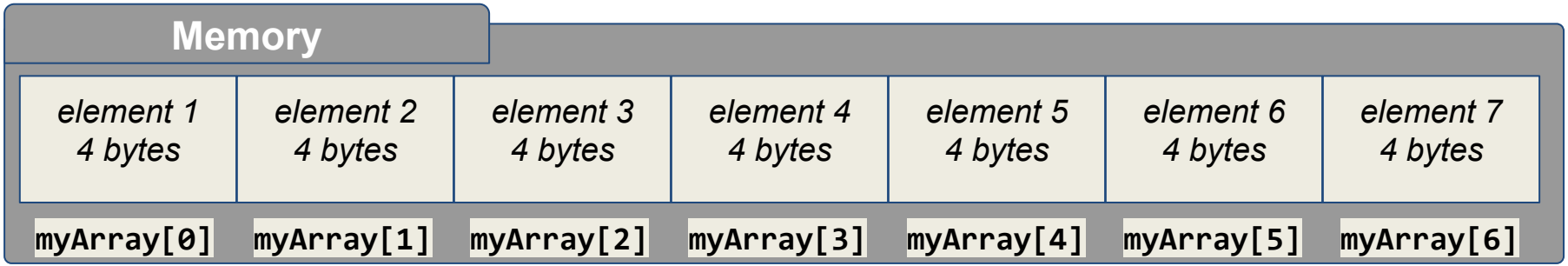
- that 4 bytes need to be reserved for each of them,
- which are organized so they span the whole numbers from -2147483648 to 2147483647
- and that **`height`** * **`width`** is a legal operation

5.1. Arrays

- An array is a serially numbered collection of variables that are all of the same *type*
- The number of elements is the *size* of the array
- Array elements are accessible via their *index*, from 0 to size-1

For example:

`float myArray[7];` is an array of 7 `float` variables, indexed from 0 to 6:



5.1. Arrays: Initialization, sizeof

- An array can be initialized by listing the elements between curly braces, **{** and **}**, and separated by commas:

```
double myArray[] = {1.09, 2.18, 4.36, 8.72};
```

  In this case, the array will automatically get the size 4

- `sizeof` is built-in operator that returns the number of *bytes* for the given variable or type:

```
int myArraySize = sizeof(myArray) / sizeof(myArray[0]);  // 16/4
```

- Loops are typically used for larger arrays:

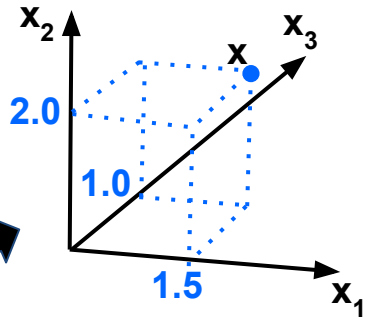
```
bool myArray[400];
for (int i = 0; i < 400; i++) myArray[i] = false;
```

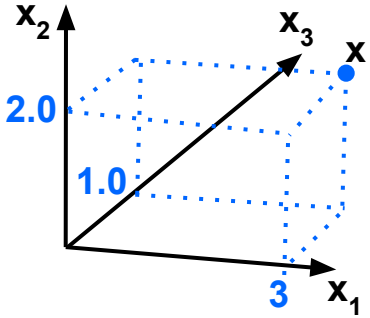
5.1. Arrays

- Example: a three-dimensional vector

```
double y[3]; // y is a 3d vector
y[0] = 1.5;
y[1] = 2.0;
y[2] = 1.0;
// or shorter:
double x[] = { 1.5, 2.0, 1.0 };

x[0] = 3.0;
```



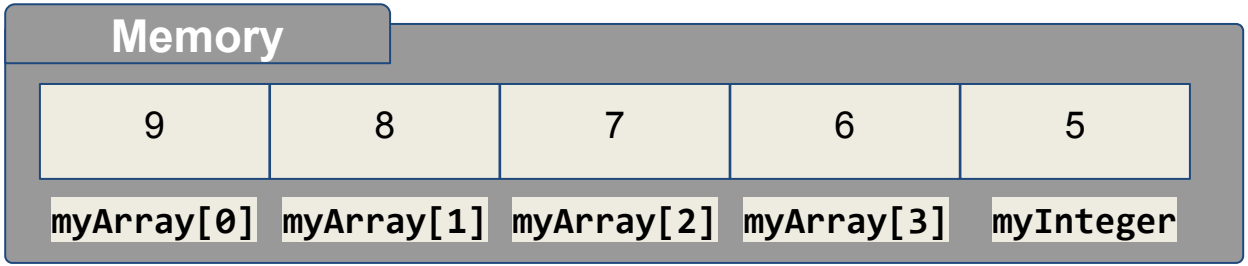


5.1. Arrays: Writing beyond the array boundary

- Most C++ compilers allow using *any* array indices to access array elements, even incorrect ones
- Non-existing array elements are usually other parts of memory, such as other variables or program code:

```
int myArray[4] = {9, 8, 7, 6};
int myInteger = 5;
std::cout << myArray[4] << std::endl;  // returns only a warning
```

- What could happen: `myArray[4]` returns the value of myInteger:



5.1. Arrays

Example 01 (difficulty level: 🌶️)

```
/**
  Write a program that initializes an array of 50 booleans, repeatedly having two
  elements with a true value, followed by one element with false.
  So the array starts with: true, true, false, true, true, false, true, true, …
  Do not use any variables other than myArray and a loop iteration variable.
*/

int main() {
  bool myArray[50];



  return 0;
}
```

5.1. Arrays

Example 02 (difficulty level: 🌶️🌶️)

```
/**
  Write a program that lets a user fill an array of 10 integers, using a loop,
  and then calculate and output the average of all given numbers to the terminal.
  Assume that the user enters a valid number each time.
  */
#include <iostream>  // to allow use of std::cout, std::cin, and std::endl
int main() {
  int myArray[10];




  return 0;
}
```

5.1. Arrays

Example 03 (difficulty level: 🌶🌶🌶)

```
/**
  Write a program that draws a histogram or bar chart through
  an array of 17 integers. To 'draw' the bars, use the string
  "\u2589" or an empty space.
  */
#include <iostream>  // std::cout, std::cin, and std::endl
#include <random>    // rand(), returns a pseudo-random int
int main() {
  int myArray[17];
  for (int i = 0; i < 17; i++) {  // fill array with random
    myArray[i] = ( rand() % 25 ); // numbers between 0 and 24
    // draw here with std::cout and std::endl
    std::cout << myArray[i] << '\n';
  }
  return 0;
}
```

example output:

```
7 ▉▉▉▉▉▉▉
24▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
23▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
8 ▉▉▉▉▉▉▉▉
5 ▉▉▉▉▉
22▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
19▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
3 ▉▉▉
23▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
9 ▉▉▉▉▉▉▉▉▉
15▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
15▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
17▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
17▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉
12▉▉▉▉▉▉▉▉▉▉▉▉
3 ▉▉▉
2 ▉▉
```

5.2. Multidimensional Arrays

- An array can be multidimensional, for example 2-dimensional:
  ```
  int myTable[2][4] = { {1, 2, 3, 4}, {5, 6, 7, 8} };
  ```
- This array is essentially an array of 2 arrays: `myTable[0]`, `myTable[1]`
- Initialization of larger arrays typically needs nested loops:
  ```
  double map[100][20];
  for (int x = 0; x < 100; x++) {
    for (int y = 0; y < 20; y++) {
      map[x][y] = 0.0;
    }
  }
  ```
- `sizeof(myTable)` will return the total size, so 2x4x4=32 bytes
- `sizeof(myTable[0])` will return 4x4 = 16 bytes

5.2. Multidimensional Arrays: Maze Game v.3.00

- Expand on version 2.00 by drawing an actual maze in the screen background, in a tiled way (since the screen can be any size)
- Add this as a two-dimensional array that you initialize yourself in the **`clearScreen`** function to build up a maze, for example:

```
int maze[][15] = { {1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
                   {1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0},
                   {1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0},
                   {1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1},
                   {1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1},
                   {0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1},
                   {1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0},
                   {1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0},
                   {1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0}
                 };  // array for drawing a maze as a background
```

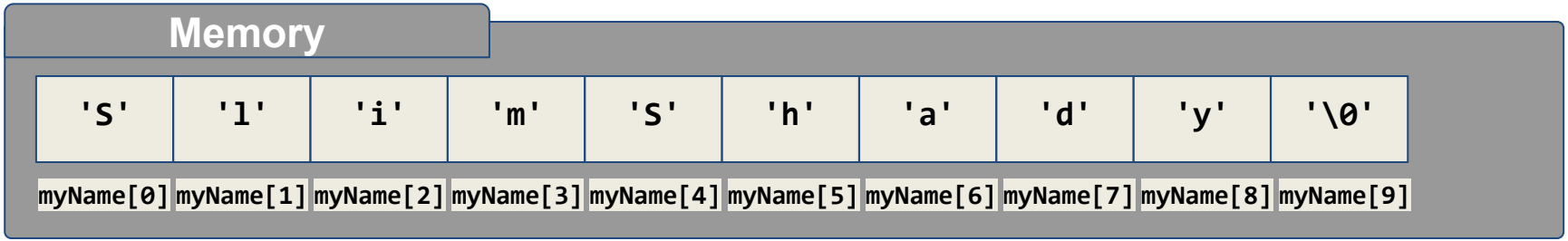
5.2. Multidimensional Arrays: Maze Game v.3.00

```
/* Third draft of Maze Game: We add an actual maze to our module "drawMaze" */
#include "drawMaze.h"  // functions related to drawing the maze and player
int main() {
  auto c = ' ';          // used for user key input
  auto x = 10, y = 10;   // (x,y) position of player: start at (10,10)
  initNCurses();         // initialize ncurses window and draw the maze
  while ( c != 'q' ) {   // as long as the user doesn't press q ..
    clearScreen();
    draw(x, y, '@', 2);   // draw our player and maze, check for collisions
    c = getch();          // capture the user's pressed key
    switch (c) {
      case 'w': y--; break;  // go up
      case 's': y++; break;  // go down
      case 'a': x--; break;  // go left
      case 'd': x++; break;  // go right
    }
  }
  endwin();           // ncurses function: close the ncurses window
}
```

5.3. Arrays: Strings (Arrays of char)

- Strings are sequences of symbols, for example to store textual data
- In C++, there is no built-in (primitive) string type. Sequences of characters can easily be implemented as an **array** of **`char`** variables, which *always* end with a zero (a character that has the value **`0`**, or also: **`'\0'`**, but NOT **`'0'`**):

```
char myName[10] = {'S', 'l', 'i', 'm', 'S', 'h', 'a', 'd', 'y', 0};
std::cout << myName << '\n';  // returns contents of myName
```



5.3. Arrays: Strings (Arrays of char)

- Constant C strings can be used for initializing:

```
char yourName[] = "Marshall Bruce Mathers III";  // works, too, and
                                                 //   ends with a 0
```

- We have already used constant strings when writing output for the terminal:

```
#include <iostream>
std::cout << "This is a string!\n";
```

- The ending zero (which also is present in the constant strings such as these two above) makes sure that we never go beyond the end of the string
- As such, the empty string `""` contains still one character (with value `0`, or also: `'\0'`, but NOT `'0'`)

5.3. Arrays: Strings (Arrays of char)

- With arrays of characters, you can manage any string already, but you will see that strings are not as easy to deal with as the basic types (`int`, `float`, `double`, `bool`, `char`). For example concatenating two strings is lots of work:

```
/** Write a program that concatenates two strings, s1 and s2, no matter
    what size they have */
#include <iostream>  // use std::cout, std::cin, and std::endl
int main() {
  char s1[] = "Apples and ", s2[] = "oranges";
  // create a new string s, which contains s1 and s2 below:


  std::cout << "Concatenated string: " << s << '\n';
}
```

5.4. Arrays as function parameters

- In C++, array parameters are passed ***by reference***

```
void swap( int a[10], int i, int j) {  // this swap function works!
  int temp = a[i];  // after this function ends, the original array a
  a[i] = a[j];      // will have swapped the values in its elements i
  a[j] = temp;      // and j. Variables i, j, and temp were created
}                   // at function start and are removed from memory
```

- The function above thus uses the actual array parameter, not a copy
- With ***call-by-reference***, variables given as actual parameters may be changed by the function
- In a function declaration, arrays can be of unspecified length:

```
void swap( int a[], int i, int j);  // Note we'll have to check for a's size
```

5.5. Reading char arrays from the terminal

- When trying our this approach:

```
char buffer[80];
std::cin >> buffer;
```

  you will see this has a few flaws: **`cin`** stops reading beyond the first whitespace character (so we cannot input sentences), and we might have a buffer overrun when we enter more than 80 characters

- The correct approach is to use:

```
char buffer[80];
std::cin.get( buffer, 80 );  // Reads at most 79 characters, 0 is last element
```

- In the above, **`get()`** seems to be a function, but: What exactly is **`cin`**?

5.6. Lambda Expressions (since C++11)

- Lambda expressions construct a **closure**: an unnamed function object that is capable of capturing variables in scope
- They are often used as callbacks (functions as arguments), for example when iterating over containers such as arrays (see also STL later)
- These are typically used for short code snippets that are not reused (they are **inline**) and do not specifically require a name:

```
auto x = [](char symbol) { std::cout << symbol << ' '; };
auto x = [](double d, int t) -> double { return (d<t)?0:d; };
```

capture clause (see next slide) parameters return type function body

5.6. Lambda Expressions (since C++11)

- We can capture external variables from the enclosing scope in three ways using the capture clause:
  - `[&]`: capture all external variables by reference
  - `[=]`: capture all external variables by value
  - `[a, &b]`: capture variable **a** by value, and variable **b** by reference

```
int a = 7, b = 14;
auto swap = [&a, &b]() -> { int t = a; a = b; b = t; };
```

- Lambdas are the simplest way of passing functions as arguments, two other methods are (1) passing functions as pointers and (2) using the **`std::function<>`** template class → see [more in-depth information] or later in this course

5.7. Range-based Loops (since C++11)

- The foreach loop or range-based for loop eases iterating over data
- It leaves out the iterator, initialization and stopping conditions:

```
#include <iostream>  // output to the console
int main() {
  int array[]= { 8, 2, 7, 2, 8, 7, 9, 1};
  for ( auto value : array ) {  // foreach loop over array
    std::cout << value << ' ';
  }
  std::cout << '\n';
}
```

5.7. Range-based Loops (since C++11)

- for multi-dimensional arrays, foreach loops look like this (using *references* → see later in chapter 7):

```
#include <iostream>  // output to the console
int main() {
  int array[][]= { {8, 2, 7}, {2, 8, 7}, {9, 1, 0} };
  for ( auto & row : array ) {       // loop over 2d array rows
    for ( auto & element : row ) {  // loop over row's elements
      std::cout << element << ' ';
    }
  }
  std::cout << '\n';
}
```

5.7. Range-based Loops (since C++11)

- Example 04 (difficulty level: 🌶️🌶️)

```
#include <iostream>  // output to the console with std::cout
int main() {
  int myArray[7][7];

  // use foreach loops to initialize myArray, so that each
  // element's value is one higher than the previous:

  // print myArray using foreach loops, one row per line,
  // zero-pad the elements if they are smaller than 10:

}
```

5.7. Range-based Loops (since C++11)

- **`std::for_each`** loops are similar to range-based for loops, and provided in **`<iostream>`**
- They apply a *function* to each of the elements in the range [first,last):

```
#include <iostream>  // output to the console, for_each
int main() {
  char array[] = {'H', 'e', 'l', 'l', 'o', '?'};
  std::for_each(std::begin(array), std::end(array),
                [](char sym) { std::cout << sym << ' '; });
  std::cout << '\n';
}
```