

# 1 Week 10

## 1.1 Functions and control flow

### 1.1.1 Passing arrays to functions

When an array is passed as a function argument, it decays to a pointer to its first element. This behavior occurs primarily in C-style arrays, where the array name acts as a pointer to its first element when used as a function argument. For example, if an array `arr` is passed as a function argument, the function receives a pointer to `arr[0]` as the parameter. The decay results in the loss of size information about the array within the function scope. Since the memory address of the array doesn't tell you anything about the original size of the array. Often when creating function with C arrays, you pass along the size as a parameter. Since you learned pointer arithmetic, now it can be performed inside the function since the array decays into a pointer. This means incrementing the pointer will point to the next element of the array.

### 1.1.2 Default values

A function with default values is a function that allows parameters to be called without ever providing arguments explicitly. When a function is defined with default parameters, those arguments can be omitted during function calls, and the default values specified in the function declaration will be used instead. The syntax is as follows:

```
type0 functionName(type1 param1 = default_value1, type2 param2 = default_value2, ...) {  
    //code  
}
```

Default values are assigned to function parameters in the function declaration. When calling the function, parameters with default values can be omitted, and the default values defined in the function declaration will be used. Default arguments are only allowed for parameters at the end of the parameter list. Once you provide a default value to any parameter, all the subsequent parameters defined to the right of it must have default values too.

### 1.1.3 Range-based for loops

Range-based for loops in C++ provide a simple way to iterate over elements in a container or sequence-like structure. The syntax is as follows:

```
for (auto& element : container) {  
    cout << element;  
}
```

The variable `element` represents a reference to each element in the `container`. The variable `container` can be any iterable container such as arrays, `vector`, `list`, `set`, etc. The `auto` keyword deduces the type of elements automatically.

The key features of a range-based for loop are:

1. Range-based for loops enhance code readability by abstracting away iterator or index details.

2. They simplify iteration over the entire container without worrying about iterators or loop termination conditions.
3. Passing by reference (`auto&`) allows for modification of elements. So then `const auto&` keeps elements constant. Using `auto` alone will create a temporary copy of the element and won't modify the original.
4. They iterate sequentially. For unordered containers, the order is undefined.

Range-based for loops are suitable for iterating through collections without index manipulation, offering cleaner syntax and reducing errors. In PIC10B/PIC16A, you'll learn about another data type called an iterator.

## 1.2 Recursion

Recursion is a programming technique where a function calls itself inside the definition. It facilitates problem solving by dividing them into smaller, more manageable parts. Recursion is implemented in code by having a function call itself to solve a smaller instance of the same problem until it reaches some condition where the function stops calling itself.

Let's consider the factorial definition in mathematics:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ n \times (n - 1)! & \text{if } n > 1 \end{cases}$$

For example,

- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- $10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3,628,800$
- $0! = 1$
- $1! = 1$

So a factorial involves taking the number and multiplying it by one less than that number. Then doing that process again until the number reaches 0 or 1. Once it reaches 0 or 1, the process stops.

```
#include <iostream>
using namespace std;

unsigned int factorial(unsigned int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

```

int main() {
    unsigned int number = 5;
    unsigned int result = factorial(number);

    cout << "Factorial of " << number << " is: " << result << endl;

    return 0;
}

```

Recursions must have a case to allow them to stop. Otherwise your computer will continue running indefinitely. In the example above, the factorial will return  $n * \text{factorial}(n - 1)$ , calling itself, until  $n$  is 0 or 1. Then it will return 1 and stop calling itself.

## 1.3 Other Data Types

In this course you've learned variables and how they can take on different types. There's the simplest, fundamental types such as integers (`int`, `unsigned int`, `size_t`), floating-point types (`float`, `double`), character types (`char`), and boolean types (`bool`). These will appear similar in other programming languages. Then you learned about more complicated types that are used as containers to hold elements such as `vectors` and `Carrays`. Classes are used to create these types and define the functions that act on instances of the classes. Today we will mention a few more classes and why they're used in C++ over something like a `vector`.

### 1.3.1 Sets

A `set` is an associative container that stores a collection of unique elements following a specific sorting order. Sets are accessed by using `#include <set>`.

The following are features of a set:

- Elements in a set are unique; duplicates are not allowed.
- Elements are sorted by value, typically in ascending order, unless a custom sorting order is specified.
- Internally, sets are implemented as red-black trees, allowing efficient insertion, deletion, and search operations. (In short, accessing things in a set is really fast).
- Unlike arrays or vectors, sets don't allow direct access using indexes; iteration through elements is used instead (range-based for loop).
- Common member functions include `insert`, `erase`, `count`, `size`, and `empty` for manipulation and querying.

Sets are desirable when you don't want duplicate elements in a collection. For small collections, accessing elements in a set can be faster than accessing elements in an array or vector.

### 1.3.2 Unordered Sets

An `unordered_set` is an associative container that stores a collection of unique elements, much like a set, but with an unordered arrangement. It's accessed by including `<unordered_set>`. Here are the main characteristics of an `unordered_set`:

- Unlike `sets` that maintain elements in a sorted order, `unordered_set` doesn't impose any specific order on its elements. Kind of like sets in mathematics.
- Elements within an `unordered_set` are unique; duplicate elements are not allowed, similar to `sets`.
- Internally, `unordered_set` is implemented using a hash table, allowing for efficient insertion, deletion, and search operations, typically with constant-time complexity.
- As the name suggests, the elements in an `unordered_set` are not sorted by value. Instead, they are stored based on their hash values, making them unordered.
- Similar to `sets`, direct indexing or accessing elements by index is not supported in `unordered_set`. You use range-based for loop to move through all the elements.
- Common member functions include `insert`, `erase`, `size`, and `empty`, which enable manipulation and querying of elements.

`unordered_set` is advantageous when uniqueness is crucial, and the order of elements is not important. It's particularly efficient when you need fast insertion, deletion, and search operations, making it suitable for scenarios where quick element retrieval is required and the order of elements is not a concern.

### 1.3.3 String Literals

A string literals are sequences of characters enclosed in double quotes (`"`). We saw that in our first lecture when we printed `"Hello, World!"` to the console. String literals are used to represent constant arrays of characters (`const char[]`) and are commonly employed to initialize or assign values to C-style strings or objects of the `std::string` class.

```
const char* cStyleString = "Hello, World!";  
// C-style string initialized with a string literal  
  
std::string cppString = "Welcome to C++!";  
// std::string object constructed with a string literal
```

String literals are null-terminated, meaning they are automatically appended with a null character (`'\0'`) to mark the end of the string. This null character distinguishes the end of the string when working with C-style strings.

It's important to note that string literals are immutable, meaning their content cannot be modified directly. Attempting to modify a string literal's content will result in undefined behavior.

## 1.4 Iterators

Iterators allow us to navigate through containers like arrays, vectors, sets, etc. and provide a way to access their elements. They act as pointers that point to elements within the container, allowing traversal and manipulation of container elements.

### Example

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    vector<int>::iterator it = vec.begin();

    while (it != vec.end()) {
        std::cout << *it << " ";
        ++it;
    }

    return 0;
}
```

This example demonstrates iterating through a `vector` using iterators. It starts from the beginning of the vector (`vec.begin()`) and moves until the end (`vec.end()`), printing each element along the way.

## 1.5 Final Review

## 1.6 Exercises

### 1.6.1 Practice coding exercises

1. Write a C++ program that defines a function to display book information. The function should have three parameters: `title`, `author`, and `year`. Set default values for `author` and `year`. Define a function `displayBookInfo()` that takes the `title`, `author`, and `year` as arguments. The default value for `author` should be “Unknown” and the default value for `year` should be 0. Implement the function to display the book information using the provided or default values.
2. In this exercise you’ll write some functions that take arrays as arguments.
  - (a) Write a function to print an array.
  - (b) Write a function that prints out `i, v[i]` times on one line.  
For example, if you give it the array `{3, 2, 1}`, the output would be:

```
000
11
2
```

If you give it the array {2, 2, 3, 1}, the output would be:

```
00
11
222
3
```

Test this function by providing the array as an argument and also creating a pointer to the first element of the array and providing that as an argument to the function.

3. There's a unproven theorem in math called the Collatz conjecture. You can take any integer greater than 1 and transform it to 1 by the following steps:

- If the number is even, divide it by 2.
- If the number is odd, multiply it by 3 and add 1.

This will generate a sequence called the Collatz sequence.

A PIC10A student begins to write their own function to print the Collatz sequence to the console. You may assume `<iostream>` has been included too.

```
int collatz(int n) {
    cout << n << ' ';

    if (n % 2 == 0){return collatz(n / 2);}
    else {return collatz(3 * n + 1);}

}
```

Currently, calling `collatz(3);` in `main` prints to the console

3 10 5 16 8 4 2 1 4 2 1 4 2 1 4 2 ...

where the sequence 1 4 2 continues seemingly forever.

To the function above, add code to prevent an infinite depth recursion. With your fix, how many times will the function be called?

4. Using recursion, write your own version of `cmath`'s function `pow`. Recall you can use to to compute the power of a number like  $2^4 = \text{pow}(2,4)$ .

5. Write a C++ program that demonstrates set operations and uses a range-based for loop to display the elements in the set.
- (a) Create two `sets` of integers: `set1` and `set2`.
  - (b) Populate `set1` with elements 1 to 5 and `set2` with elements 3 to 7.
  - (c) Now create the following sets:
    - `unionSet` contains all elements from both `set1` and `set2`.
    - `intersectSet` contains elements common to both `set1` and `set2`.
    - `diffSet` contains elements that are in `set1` but not in `set2` and that are in `set2` but not in `set1`.
  - (d) Display each resulting set using a range-based for loop.

### Sample Output:

```
Elements in unionSet: 1 2 3 4 5 6 7
Elements in intersectSet: 3 4 5
Elements in diffSet: 1 2 6 7
```

## 1.6.2 Predict the output exercises

Write the console outputs (not including the Program ended with exit code: 0 line). Assume the code runs error-free, includes the appropriate headers, and is using `std` namespace.

```
6. #include <iostream>
using namespace std;

void greet(string name = "you", int age = 0) {
    cout << "Hello, " << name << "! You are " << age << " years old." << endl;
}

int main() {

    greet();
    greet("Alice");
    greet("Bob", 30);

    return 0;
}
```

```
7. #include <iostream>
using namespace std;
```

```

int main() {
    int x = 10;
    int* p = &x;
    cout << p << endl; // Let's say its 0x16fdff268
    p++;
    cout << p << endl;
    p++;
    cout << p << endl;
    p+=4;
    cout << p << endl;
    return 0;
}

```

8. `#include <iostream>`  
`using namespace std;`

```

int main() {
    int x[] = {2, 4, 6, 8};
    int y = 7;
    int* z = &y;

    *z *= *(x + 1);
    cout << y << endl;

    *(x + 2) += *z;
    cout << x[0] << ' ' << x[1] << ' ' << x[2] << ' ' << x[3] << endl;

    return 0;
}

```

9. `#include <iostream>`  
`#include <set>`  
`using namespace std;`

```

int main() {
    set<int> mySet;
    mySet.insert(5);
    mySet.insert(3);
    mySet.insert(7);

    cout << mySet.size() << endl;

    cout << mySet.count(3) << endl;
}

```

```

mySet.erase(3);

cout << mySet.count(3) << endl;

cout << boolalpha << mySet.empty() << endl;

return 0;
}

```

```

10. #include <iostream>
    #include <set>
    #include <vector>
    using namespace std;

    int main() {
        vector<int> input = {2, 5, 3, 7, 2, 8, 5, 3};

        set<int> s;
        vector<int> v;

        for (int num : input) {
            s.insert(num);
        }

        for (int num : input) {
            v.push_back(num);
        }

        cout << "Set: ";
        for (int num : s) {
            cout << num << " ";
        }
        cout << endl;

        cout << "Vector: ";
        for (int num : v) {
            cout << num << " ";
        }
        cout << endl;

        return 0;
    }

```

```

11. #include <iostream>
    using namespace std;

    int main() {
        const char a[] = "Donut";
        cout << sizeof(a) << endl;

        return 0;
    }

```

```

12. #include <iostream>
    using namespace std;

    int main() {
        const int a[] = {1, 2, 3};
        cout << sizeof(int) << endl;
        cout << sizeof(a) << endl;

        return 0;
    }

```

### 1.6.3 Error spotting exercises

Error spotting will be on the exam. Look at the follow code below and state:

- What errors occurred (build, run time, or logic) or if there are none.
- In the case of a logic error, what does the code currently accomplish?
- How the code can be fixed (use the line numbers to help with the description).

A description of how the code should perform will be put in a block comment. After you make your initial guess, type the code in your IDE and run it. Copying very carefully may help you find some of the errors. Play around with the code to fix it.

```

13. 1  #include <iostream>
    2  using namespace std;
    3
    4  /* The user wants to calculate the area (12) of a rectangle with
    5  the length 3 and width 4. */
    6  double calculateArea(double length, double width = 5.0) {
    7      return length * width;

```

```

8   }
9
10  int main() {
11      double l = 3.0, w = 4.0;
12      cout << "Area: " << calculateArea(l) << endl;
13      return 0;
14  }

```

14.

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  /* The user wants to print the number 5 to the console.
6     Otherwise, they want "Hello" to print. */
7  void displayMessage(string msg = "Hello") {
8      cout << msg << endl;
9  }
10
11  int main() {
12      displayMessage(5);
13      return 0;
14  }

```

15.

```

1  #include <iostream>
2  using namespace std;
3
4  /* This code should print the 'Number: 10'. */
5  class MyClass {
6  public:
7      void printMessage(int num = 10) {
8          cout << "Number: " << num << endl;
9      }
10 };
11
12 int main() {
13     MyClass obj;
14     obj.printMessage();
15     return 0;
16 }

```

```

16. 1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code is meant to print '1 and 5' to the console. */
    5  void printValues(int a = 1, int b) {
    6      cout << "Values: " << a << " and " << b << endl;
    7  }
    8
    9  int main() {
   10      printValues(5);
   11      return 0;
   12  }

```

```

17. 1  #include <iostream>
    2  using namespace std;
    3
    4  /* This code is meant to take the elements of
    5     an array and double them. */
    6  void double_it(int arr, int size) {
    7      for (int i = 0; i < size; ++i) {
    8          arr[i] *= 2; // Double each element of the array
    9      }
   10  }
   11
   12  int main() {
   13      int arr[] = {1, 2, 3, 4, 5};
   14      int size = sizeof(arr) / sizeof(arr[0]);
   15      double_it(arr, size);
   16      return 0;
   17  }

```

```

18. 1  #include <iostream>
    2  #include <set>
    3  using namespace std;
    4
    5  /* The user is printing out the elements of s. */
    6  int main() {
    7      set<int> s = {5, 2, 8, 3, 1};
    8
    9      for (size_t i = 0; i < s.size(); ++i) {
   10          cout << s[i] << endl;
   11      }

```

```

12
13     return 0;
14 }

19. 1  #include <iostream>
    2  #include <set>
    3  using namespace std;
    4
    5  /* The following code is meant to go through and capitalize all
    6  the chars in the set s. Then print them out to see 'BAT'. */
    7  int main() {
    8      set<char> s = {'b', 'a', 't'};
    9
   10      for (char val : s) {
   11          val += 'A' - 'a';
   12      }
   13
   14      for (char val : s) {
   15          cout << val;
   16      }
   17      cout << endl;
   18
   19      return 0;
   20 }

20. 1  int main() {
    2      const char* s1 = "Hello, world!";
    3      cout << s1[0] << s1[1] << "..." << s1[11] << s1[12] << endl;
    4      cout << (s1[13] == '\0') << endl;
    5
    6
    7      s1 = "bye!";
    8      cout << s1[1] << s1[2] << s1[3] << s1[4] << endl;
    9
   10      s1[0] = 'B';
   11      cout << s1[1] << endl;
   12
   13      return 0;
   14 }

```

```

21. 1  #include <iostream>
    2  #include <vector>
    3  using namespace std;
    4
    5  int f() {
    6      int i = 1;
    7      return i + f();
    8  }
    9
   10  int main() {
   11
   12      cout << f() << endl;
   13
   14      return 0;
   15  }

```

22. ( The all\_caps function is fine. )

```

    1  #include <iostream>
    2  #include <string>
    3  using namespace std;
    4
    5  /**
    6   Capitalizes all letters in a string that are lowercase.
    7   @param str string to capitalize.
    8   @return All caps string.
    9   */
   10  string all_caps(string& str) {
   11      for (int idx = 0; idx < str.length(); idx++) {
   12          if (str[idx] >= 'a' && str[idx] <= 'z') {
   13              str[idx] -= 'a' - 'A';
   14          }
   15      }
   16      return str;
   17  }
   18
   19  /**
   20   Yells a name.
   21   @param first The first name.
   22   @param last The last name.
   23   */
   24  void yell_name(string& first, string& last) {
   25      cout << all_caps(first) << '!' << ' ' << all_caps(last) << '!' << endl;
   26  }

```

```
27
28  /* This code is meant to print:
29    BART! SIMPSON!
30  */
31  int main() {
32      yell_name("Bart", "Simpson");
33      return 0;
34  }
```