Virtual Environment Development

A (Soft) Introduction to C++, Part I

Vassilis Markos, Mediterranean College, Fall 2024 – 2025

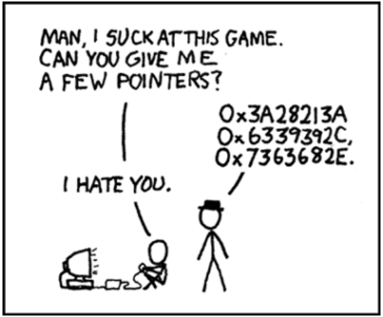
Week 06

Contents

1. Why C++?
2. C++ Functions
3. Pointers
4. Arrays
5. 2D Arrays
6. Strings
7. Fun Time!

Why C++?

Some Programming Humour, First...



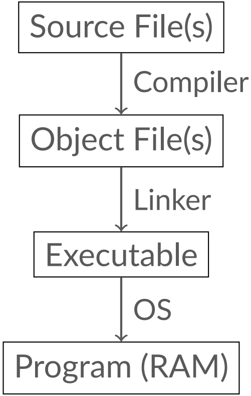
If you already understand this, you might be in the wrong room. Source: `https://xkcd.com/138/`.

Why Even Use C++?

- C++ is higher level than assembly (okay, Python is too...).
- C++ is easily maintainable (Python is much better in that regard...).
- C++ is highly versatile / portable (Python is too...).
- C++ allows for low–level handling of memory allocation which:
    - facilitates concurrent programming (Python has readymade libraries for that).
    - is quite useful in terms of **studying and understanding the core principles of concurrent programming** (okay, you win).
- C++ is really fast and a nice–to–have skill (Python is nice to know, too, but not so fast).

The Compilation Process (in general)

- **Source File:** The code file we write on a PC.
- **Object File:** The compiled code file, usually machine / CPU instructions (e.g., assembly).
- **Executable:** A file that can be executed by a certain Operating System (OS).
- **Compiler:** A computer program that transforms a source file to an object file.
- **Linker:** A computer program that interconnects a source file with other required existing programs (libraries).

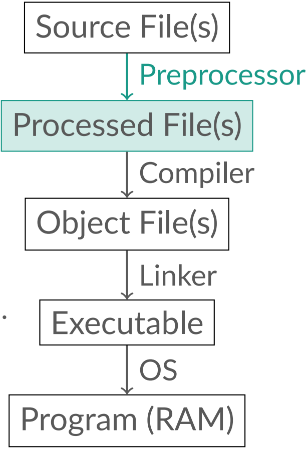


The C++ Compilation Process

C++ adds one more step in the compilation process, the **Preprocessor**, which is responsible for:

- interpreting certain directives, e.g., those starting with `#`, and;
- generating a single code file to be fed to the compiler.

For more: `https://en.cppreference.com/w/cpp/preprocessor`.



A Note on C++ Compilers

To use C++ on a PC, you need to install an OS–compatible compiler:

- For Windows (and, hence, this course), you will need a Windows–compatible compiler. The most popular choice is the **Visual C++ compiler** by MS, which comes with Visual Studio.
  - Instructions for installing C++ alongside the entire Visual Studio suite: `https://visualstudio.microsoft.com/vs/features/cplusplus/`.
  - For minimalists, instructions including only C++ and VSCode: `https://code.visualstudio.com/docs/languages/cpp`.
- For Linux / UNIX users, installing `gcc`, which includes `g++`, is easier: `https://gcc.gnu.org/`.

Heads up!

All C++ Compilers ARE NOT THE SAME!

Be careful when installing a C++ compiler, as there are subtle differences when it comes to what is being considered valid C++ code.

- In general, `gcc` is more relaxed compared to Visual C++.
- However, Visual C++ is more widely used (due to MS) and we will stick to its specification of C++.
- Visual C++ documentation: `https://github.com/MicrosoftDocs/cpp-docs/tree/main/docs`.
- Probably the most important difference is that `g++` allows variable length arrays (as we will see soon) while Visual C++ does not.

helloWorld.cpp

```cpp
// source/helloWorld.cpp

#include <iostream> // Loads I/O functionality

int main() { // Main signature (it returns an integer)
    std::cout << "Hello, world!\n"; // Prints to out stream
    return 0; // Mandatory, since main() returns an integer.
}
```

Notes on `helloWorld.cpp`

- `#include` is a Preprocessor command informing the preprocessor that it should “copy-paste” into this file the contents of the `iostream` library.
- `std` is a **namespace**, i.e., a set of commands accessible under that name.
- `::` is the **scope resolution operator**, i.e., it tells C++ where to look for a certain command. So, in this case, we tell C++ to look for `cout` in the `std` namespace.
- `return 0;` is a typical spell included in the end of every `main()` function.

Heads up!

Namespaces

We can inform the compiler to look for any undefined directives in a certain namespace, as shown below. The following version of `helloWorld.cpp` is equivalent to the one above, but more concise.

```cpp
// source/helloWorldUsingNS.cpp

#include <iostream> // Loads I/O functionality
using namespace std;

int main() { // Main signature (it returns an integer)
    cout << "Hello, world!\n"; // Prints to out stream
    return 0; // Mandatory, since main() returns an integer.
}
```

Fancy String Characters in C++

Sequence	Meaning
`\a`	System bell (“beeps”)
`\b`	Backspace
`\f`	Page break (form feed)
`\n`	Line break (newline)
`\r`	Carriage return (returns the cursor to start of line)
`\t`	Tab
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\c`	Character represented by integer c

Frequently Used C++ Data Types

Type Name	Description	Size
`char`	Single text character, indicated with single quotes	1 byte
`int`	Signed or unsigned integer	4 bytes
`bool`	Boolean	1 byte
`float`	Float number, 7 decimal digits accuracy	4 bytes
`double`	Double accuracy float, 15 decimal digits.	8 bytes

Frequently Used C++ Operators

Operator(s)	Description
`+`, `-`, `*`, `/`	Addition, subtraction, multiplication, division, priority determined as in maths (plus parentheses).
`%`	Modulus operator: Remainder of integer division, e.g., `13 % 4` evaluates to `1`.
`&&`, `||`, `!`	Logical `AND`, `OR`, `NOT`.
`==`, `!=`	`equals` and `not equals`.
`<`, `>`	`larger than` and `less than`.

Flow Control in C++

```cpp
// source/flowControl.cpp
#include <iostream>
using namespace std;

int main() { // Main signature (it returns an integer)
    char knowsCpp; // Declaring a char variable
    cout << "Do you know C++? (y/n)\n";
    cin >> knowsCpp; // Read from in stream
    if (knowsCpp == 'y') { // Base case
        cout << "Congrats! You already know C++!\n";
    } else if (knowsCpp == 'n') { // If the above fails
        cout << "Dont't worry, you can attend this course!\n";
    } else { // In case all of the above have failed
        cout << "Please, enter 'y' or 'n'.\n";
    }
    return 0;
}
```

`while` Loops in C++

```cpp
// source/whileLoop.cpp
#include <iostream>
using namespace std;

int main() {
    const int GUESS = 4; // Constant value (immutable)
    int x; // Declare variable
    cout << "Guess what I'm thinking (int): ";
    cin >> x;
    while (x != GUESS) { // Repeat while condition holds
        cout << "No, try again: ";
        cin >> x;
    }
    cout << "Congratulations! You guessed right!\n";
    return 0;
}
```

`for` Loops in C++

```cpp
// source/forLoop.cpp
#include <iostream>
using namespace std;

int main() {
    int x; // Declare variable
    int sum = 0; // Initialise variable
    for (int i = 0; i < 5; i++) {
    // for (index, terminating condition; step)
        cout << "Please, enter an integer: ";
        cin >> x;
        sum = sum + x;
    }
    cout << "Sum: " << sum << ".\n";
    // We can chain outputs in cout with "<<"
    return 0;
}
```

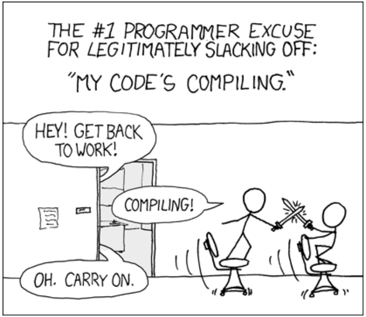
Useful Resources

Some resources you might find useful in your C++ journey:

- C++ Tutorial: `https://cplusplus.com/doc/tutorial/`
- MIT C++ Course (we will follow parts of it): `https://ocw.mit.edu/courses/6-096-introduction-to-c-january-iap-2011/`
- Visual C++ docs: `https://github.com/MicrosoftDocs/cpp-docs/tree/main/docs`.

C++ Functions

Compiling Stuff...



Breaking news from Developers' Land. Source: `https://xkcd.com/303/`.

Writing Some Code

What does this C++ code compute?

```cpp
// source/someCode.cpp
#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Please, enter a positive integer: ";
    cin >> n;
    int i = n;
    int f = 1;
    while (i > 0) {
        f *= i;
        i--;
    }
    cout << n << "! == " << f << endl;
}
```

How Can We Repeat This Twice?

```cpp
// source/someNaiveCode.cpp
#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Please, enter a positive integer: ";
    cin >> n;
    int i = n;
    int f = 1;
    while (i > 0) {
        f *= i;
        i--;
    }
    cout << n << "! == " << f << endl;
    cout << "Please, enter a positive integer: ";
    cin >> n;
    i = n;
    f = 1;
    while (i > 0) {
        f *= i;
        i--;
    }
    cout << n << "! == " << f << endl;
}
```

What About Loops?

```cpp
// source/whatAboutLoops.cpp
#include <iostream>
using namespace std;

int main() {
    int n;
    int i;
    int f;
    for (int j = 0; j < 5; j++) {
        cout << "Please, enter a positive integer: ";
        cin >> n;
        i = n;
        f = 1;
        while (i > 0) {
            f *= i;
            i--;
        }
        cout << n << "! == " << f << endl;
    }
}
```

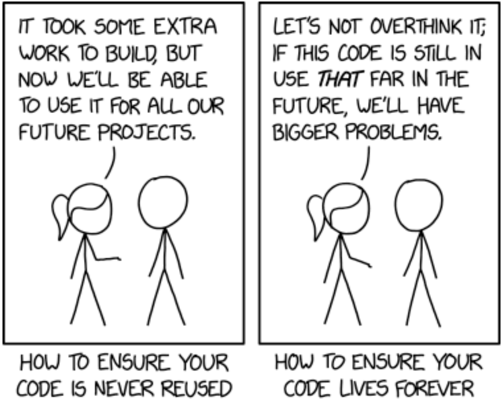
Abstraction Through Functions

```cpp
// source/functionAbstraction.cpp
#include <iostream>
using namespace std;

int factorial(int n) {
    int i = n;
    int f = 1;
    while (i > 0) {
        f *= i;
        i--;
    }
    return f;
}

int main() {
    int n;
    int f;
    for (int j = 0; j < 5; j++) {
        cout << "Please, enter a positive integer: ";
        cin >> n;
        f = factorial(n);
        cout << n << "! == " << f << endl;
    }
}
```

Why Use Functions?

- **Code Maintenance:** It is way easier to maintain your code if all functionality that is intended to be reused is defined and kept at one place.
- **Code Distribution:** Imagine reading a cryptic project where everything is defined at difficult to spot places. Avoid this for your projects as much as possible.
- **Debugging:** The more concise your code the easiest to understand and, consequently, the easiest to debug.

A Brief Reality Check


IT TOOK SOME EXTRA WORK TO BUILD, BUT NOW WE'LL BE ABLE TO USE IT FOR ALL OUR FUTURE PROJECTS.
HOW TO ENSURE YOUR CODE IS NEVER REUSED
LET'S NOT OVERTHINK IT; IF THIS CODE IS STILL IN USE THAT FAR IN THE FUTURE, WE'LL HAVE BIGGER PROBLEMS.
HOW TO ENSURE YOUR CODE LIVES FOREVER

Another day in office. Source: `https://xkcd.com/2730/`

Function Terms

- **Function signature:** The part of the function definition that includes:
  - Its name, which may be any alhpanumeric string starting with a letter.
  - Its return type, which is any C++ valid type.
  - Its arguments, which is a list of typed variable names.
- **Return value:** Every function can return up to one variable, whose type should match the function's return type.

What’s Wrong Here?

Execute the following program. What goes wrong?

```cpp
// source/problem_001.cpp
#include <iostream>
using namespace std;

double foo(int x) {
    double y = x / 2;
    return y;
}

int main() {
    cout << foo("5") << endl;
}
```

What’s Wrong Here?

Execute the following program. What goes wrong?

```cpp
// source/problem_002.cpp
#include <iostream>
using namespace std;

double foo(int x) {
    return "f";
}

int main() {
    cout << foo(5) << endl;
}
```

What’s Not Wrong Here?

Execute the following program. Why does it work?

```cpp
// source/problem_003.cpp
#include <iostream>
using namespace std;

double foo(int x) {
    return 'f';
}

int main() {
    cout << foo(5) << endl;
}
```

What’s Wrong Here?

Execute the following program. What goes wrong?

```cpp
// source/problem_004.cpp
#include <iostream>
using namespace std;

void foo(int x) {
    return x + 4;
}

int main() {
    cout << foo(12) << endl;
}
```

What’s Not Wrong Here?

Execute the following program. Why does it work?

```cpp
// source/problem_005.cpp
#include <iostream>
using namespace std;

double foo(int x) {
    double y = x / 2;
    return y;
}

int main() {
    cout << foo(6.8) << endl;
}
```

Important Notes On Functions

- Functions that do not return a value should be declared as `void`.
- C++ will try to cast types whenever it can. For instance, C++ can cast doubles to integers, so it will, if prompted to.
  - We will discuss about all of that stuff in more detail throughout this course.
  - If you feel an urge to learn more, you can look up `rvalue` and `lvalue`.
- Always remember that characters in C++ can also be treated as integers, with its pros and cons.

Will This Work?

Will this work? Answer before executing the program first!

```cpp
// source/foo.cpp
#include <iostream>
using namespace std;

int foo(int x) {
    return x + 5;
}

void foo(char x) {
    cout << x << endl;
}

int foo(double x) {
    return x + 5.06;
}

int main() {
    cout << foo(6) << endl;
    foo('3');
    cout << foo(5.8) << endl;
}
```

Function Overloading

- C++ allows us to define functions so long as they have a different signature.
- This means that two functions sharing the same name should either accept different types / number of arguments and / or have a different return type.
- Thus, we can define a function that showcases different behaviour depending on the types of its arguments. This allows for simpler and more concise APIs.

Pointers

Stars And C++

What will this program print? (Do not execute it!)

```cpp
// source/stars_001.cpp
#include <iostream>
using namespace std;

int main() {
    double x;
    cout << "Enter a double: ";
    cin >> x;
    double *y = &x;
    cout << (*y == x) << endl;
}
```

What Are * and &?

- What is `*y` in the above program at line 9?

What Are * and &?

- What is *y in the above program at line 9?
    - *y denotes the memory location of a double variable, x in our case. Such variables are called **pointers**.

What Are * and &?

- What is `*y` in the above program at line 9?
    - `*y` denotes the memory location of a `double` variable, `x` in our case. Such variables are called **pointers**.
- What is `*y` in the above program, at line 10?

What Are * and &?

- What is `*y` in the above program at line 9?
  - `*y` denotes the memory location of a `double` variable, `x` in our case. Such variables are called **pointers**.
- What is `*y` in the above program, at line 10?
  - `*y` denotes the content at the memory location stored by pointer `y`. Retrieving a pointer's pointed value is often called **dereferencing**.

What Are * and &?

- What is `*y` in the above program at line 9?
  - `*y` denotes the memory location of a `double` variable, `x` in our case. Such variables are called **pointers**.
- What is `*y` in the above program, at line 10?
  - `*y` denotes the content at the memory location stored by pointer `y`. Retrieving a pointer's pointed value is often called **dereferencing**.
- What is `&x` in the above program at line 9?

What Are * and &?

- What is `*y` in the above program at line 9?
  - `*y` denotes the memory location of a `double` variable, `x` in our case. Such variables are called **pointers**.
- What is `*y` in the above program, at line 10?
  - `*y` denotes the content at the memory location stored by pointer `y`. Retrieving a pointer's pointed value is often called **dereferencing**.
- What is `&x` in the above program at line 9?
  - `&x` indicates the memory location where the content of variable `x` is stored in the computer's memory, i.e., it is a **reference** of `x`.

Can You Predict The Output?

```cpp
// source/stars_002.cpp
#include <iostream>
using namespace std;

int main() {
    int x;
    cout << "Enter an int: ";
    cin >> x;
    int *y = &x;
    cout << *y << endl;
}
```

Can You Predict The Output?

```cpp
// source/stars_003.cpp
#include <iostream>
using namespace std;

int main() {
    int x;
    cout << "Enter an int: ";
    cin >> x;
    int *y = &x;
    cout << *( &x ) << endl;
}
```

Can You Predict The Output?

```cpp
// source/stars_004.cpp
#include <iostream>
using namespace std;

int main() {
    int x;
    cout << "Enter an int: ";
    cin >> x;
    int *y = &x;
    cout << y << endl;
}
```

Can You Predict The Output?

```cpp
// source/stars_005.cpp
#include <iostream>
using namespace std;

int main() {
    int x;
    cout << "Enter an int: ";
    cin >> x;
    int *ptrx = &x;
    (*ptrx)++;
    cout << x << endl;
}
```

Can You Predict The Output?

```cpp
// source/stars_006.cpp
#include <iostream>
using namespace std;

int main() {
    int x, y;
    cout << "Enter two ints: ";
    cin >> x;
    cin >> y;
    int *ptrx = &x;
    int *ptry = &y;
    ptry = ptrx;
    (*ptrx)--;
    ptry = &y;
    cout << x << ", " << y << endl;
}
```

Can You Predict The Output?

```cpp
// source/stars_007.cpp
#include <iostream>
using namespace std;

int main() {
    int x, *y;
    cout << "Enter an int: ";
    cin >> x;
    y = &x;
    (*y)++;
    cout << x << endl;
}
```

A const Interlude

Consider the program shown right. Which of the following inserted in line 12 will raise an error?

1. `(*p1)++;`
2. `(*p2)++;`
3. `p1 = &y;`
4. `p2 = &y;`

```
 1  // source/stars_008.cpp
 2  #include <iostream>
 3  using namespace std;
 4
 5  int main() {
 6      int x, y;
 7      cout << "Enter two ints: ";
 8      cin >> x;
 9      cin >> y;
10      const int *p1 = &x;
11      int * const p2 = &x;
12      // Enter your code here...
13  }
```

Pointers To Constant Variables

In this case, pointer p1 is pointing to a **constant** integer variable. This means that at the memory location p1 is pointing to we can make no modifications!

```cpp
// source/stars_009.cpp
#include <iostream>
using namespace std;

int main() {
    int x, y;
    cout << "Enter two ints: ";
    cin >> x;
    cin >> y;
    const int *p1 = &x;
    int * const p2 = &x;
    (*p1)++;
}
```

Pointers To Constant Variables

In this case, pointer p2 is a **constant** pointer pointing to an integer variable. This means that at the memory location p2 is pointing to we can make any modifications we want to. What we can't change is the value of the pointer itself.

```cpp
// source/stars_010.cpp
#include <iostream>
using namespace std;

int main() {
    int x, y;
    cout << "Enter two ints: ";
    cin >> x;
    cin >> y;
    const int *p1 = &x;
    int * const p2 = &x;
    (*p2)++;
}
```

All `const`?

What about this one?

```
const int * const ptr;
```

In this case:

All `const`?

What about this one?

```
const int * const ptr;
```

In this case:

- We cannot change where the pointer points to (rightmost `const`).

All `const`?

What about this one?

```
const int * const ptr;
```

In this case:

- We cannot change where the pointer points to (rightmost `const`).
- We cannot change the content of the memory location the pointer points to (leftmost `const`).

Arrays

Arrays In C++

Can you guess what the following will print?

```cpp
// source/arrays_001.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[3];
    arr[0] = 4;
    arr[1] = 6;
    arr[2] = -5;
    for (int i = 0; i < 3; i++) {
        cout << "arr[" << i << "] == " << arr[i] << endl;
    }
}
```

Array Initialisation

We can also provide array elements all at once, as follows:

```cpp
// source/arrays_002.cpp
#include <iostream>
using namespace std;

int main() {
    int arr1[5] = { 4, -2, 0, 4, 6 };
    int arr2[] = { 6, 5, 7, 9 };
    cout << "arr1[3] == " << arr1[3] << "\narr2[1] == " << arr2[1] << endl;
}
```

Dynamic Initialisation

We can also initialise the values of an array based on others' input (e.g., users, another process):

```cpp
// source/arrays_003.cpp
#include <iostream>
using namespace std;

int main() {
    char arr[3];
    for (int i = 0; i < 3; i++) {
        cout << "Please, enter a character: ";
        cin >> arr[i];
    }
    cout << arr[0] << arr[1] << arr[2] << endl;
}
```

What Will This Print?

```cpp
// source/arrays_004.cpp
#include <iostream>
using namespace std;

int main() {
    char arr[];
    for (int i = 0; i < 3; i++) {
        cout << "Please, enter a character: ";
        cin >> arr[i];
    }
    cout << arr[0] << arr[1] << arr[2] << endl;
}
```

Dynamic Initialisation And Array Size

The above must have printed something along the following lines:

```
arrays_004.cpp: In function ‘int main()’:
arrays_004.cpp:6:10: error: storage size of ‘’arr’
isnt known
6 |      char arr[];
  |           ^~~
```

This actually means that in order to **refer to an array’s element by its index** you must **first determine the array’s size!**

Pointers And Arrays

What will the following print?

```cpp
// source/arrays_005.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 5, 1};
    int* ptr = arr;
    ptr++;
    cout << *ptr << endl;
}
```

Heads up!

Pointers And Arrays

Do you observe something strange in the following?

```
int main() {
    int arr[] = {2, 6, 5, 1};
    int* ptr = arr;
    ptr++;
    cout << *ptr << endl;
}
```

Heads up!

Pointers And Arrays

Do you observe something strange in the following?

```
1 int main() {
2     int arr[] = {2, 6, 5, 1};
3     int* ptr = arr;
4     ptr++;
5     cout << *ptr << endl;
6 }
```

- **Line 3:** We declare an integer pointer and store the **array** there, **not a reference!**

Heads up!

Pointers And Arrays

Do you observe something strange in the following?

```cpp
1 int main() {
2     int arr[] = {2, 6, 5, 1};
3     int* ptr = arr;
4     ptr++;
5     cout << *ptr << endl;
6 }
```

- **Line 3:** We declare an integer pointer and store the **array** there, **not a reference!**
- Why does it work?

Heads up!

Pointers And Arrays

- In C++, arrays of type `<T>` are actually pointers to items of type `<T>`.
- This means that, when declaring an array, we are actually declaring a pointer to the first memory location occupied by its first element.
- So, arrays are actually of **pointer type!**
- This means that using `&` to get their memory address is of no use, since they already represent a memory address.

Pointer Tricks

What will the following print?

```cpp
// source/arrays_006.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 5, 1};
    float* ptr = (float*) arr;
    ptr++;
    cout << *ptr << endl;
}
```

Pointer Tricks

What will the following print?

```cpp
// source/arrays_007.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 0, 0, 4};
    double* ptr = (double*) arr;
    ptr++;
    cout << *ptr << endl;
}
```

Pointer Tricks

What will the following print?

```
// source/arrays_008.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    double* ptr = (double*) arr;
    ptr++;
    cout << *ptr << endl;
}
```

Looping Over An Array

```cpp
// source/arrays_009.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    for (int i = 0; i < 5; i++) {
        cout << "arr[" << i << "] == " << arr[i] << endl;
    }
}
```

Looping Over An Array With Pointers

Can you loop over the same array without using the `arr[i]` syntax?

Looping Over An Array With Pointers

Can you loop over the same array without using the `arr[i]` syntax?

```cpp
// source/arrays_010.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    for (int i = 0; i < 5; i++) {
        cout << "arr[" << i << "] == " << *(arr + i) << endl;
    }
}
```

Pointer Arithmetic (Again)

- In general, the expression `pointer + integer` is interpreted as: increment the `pointer` by the size of its pointing type times the `integer`.
- So, for an `int* ptr`, `ptr + 6` should be interpreted as “move the pointer `ptr` by `6 * sizeof(int)`, i.e., `6 * 4` bytes”.
- So, for a `double* ptr`, `ptr + 5` should be interpreted as “move the pointer `ptr` by `5 * sizeof(double)`, i.e., `5 * 8` bytes”.

Passing Arrays To Functions

What will this print?

```cpp
// source/arrays_011.cpp
#include <iostream>
using namespace std;

void printArray(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        cout << "arr[" << i << "] == " << *(arr + i) << endl;
    }
}

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    printArray(arr, 5);
}
```

Passing Arrays To Functions

What will this print?

```cpp
// source/arrays_012.cpp
#include <iostream>
using namespace std;

void printArray(int* arr, int length) {
    for (int i = 0; i < length; i++) {
        cout << "arr[" << i << "] == " << *(arr + i) << endl;
    }
}

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    printArray(arr, 5);
}
```

Passing Arrays To Functions

What will this print?

```cpp
// source/arrays_013.cpp
#include <iostream>
using namespace std;

void foo(int* arr, int length) {
    for (int i = 0; i < length; i++) {
        if (*(arr + i) == 0) {
            *(arr + i) = 4;
        }
    }
}

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    cout << arr[2] << endl;
    foo(arr, 5);
    cout << arr[2] << endl;
}
```

Array Decay

- A common C++ catch–phrase is that “arrays decay into pointers”.
- This simply means that, whenever required, arrays are interpreted as pointers, as we have already discussed above.
- As a consequence, when passing an array to a function, we are actually passing a pointer.
- This means that an array is always **passed by reference**. So, in case we need to pass an array be value, we have to devise various tricks we shall see in upcoming lectures.

2D Arrays

(Reminder) Arrays In C++

Can you guess what the following will print?

```cpp
// source/arrays_001.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[3];
    arr[0] = 4;
    arr[1] = 6;
    arr[2] = -5;
    for (int i = 0; i < 3; i++) {
        cout << "arr[" << i << "] == " << arr[i] << endl;
    }
}
```

(Reminder) Array Initialisation

We can also provide array elements all at once, as follows:

```cpp
// source/arrays_002.cpp
#include <iostream>
using namespace std;

int main() {
    int arr1[5] = { 4, -2, 0, 4, 6 };
    int arr2[] = { 6, 5, 7, 9 };
    cout << "arr1[3] == " << arr1[3] << "\narr2[1] == " <<
   arr2[1] << endl;
}
```

(Reminder) Dynamic Initialisation

We can also initialise the values of an array based on others’ input (e.g., users, another process):

```cpp
// source/arrays_003.cpp
#include <iostream>
using namespace std;

int main() {
    char arr[3];
    for (int i = 0; i < 3; i++) {
        cout << "Please, enter a character: ";
        cin >> arr[i];
    }
    cout << arr[0] << arr[1] << arr[2] << endl;
}
```

(Reminder) Passing Arrays To Functions

What will this print?

```cpp
// source/arrays_011.cpp
#include <iostream>
using namespace std;

void printArray(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        cout << "arr[" << i << "] == " << *(arr + i) << endl;
    }
}

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    printArray(arr, 5);
}
```

(Reminder) Passing Arrays To Functions

What will this print?

```cpp
// source/arrays_012.cpp
#include <iostream>
using namespace std;

void printArray(int* arr, int length) {
    for (int i = 0; i < length; i++) {
        cout << "arr[" << i << "] == " << *(arr + i) << endl;
    }
}

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    printArray(arr, 5);
}
```

(Reminder) Passing Arrays To Functions

What will this print?

```cpp
// source/arrays_013.cpp
#include <iostream>
using namespace std;

void foo(int* arr, int length) {
    for (int i = 0; i < length; i++) {
        if (*(arr + i) == 0) {
            *(arr + i) = 4;
        }
    }
}

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    cout << arr[2] << endl;
    foo(arr, 5);
    cout << arr[2] << endl;
}
```

(Reminder) Array Decay

- A common C++ catch–phrase is that “arrays decay into pointers”.
- This simply means that, whenever required, arrays are interpreted as pointers, as we have already discussed above.
- As a consequence, when passing an array to a function, we are actually passing a pointer.
- This means that an array is always **passed by reference**. So, in case we need to pass an array be value, we have to devise various tricks we shall see in upcoming lectures.

Declaring 2D Arrays

```cpp
// source/2d_arrays_001.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[2][3]; // Declare "rows" and "columns"
    arr[0][0] = 2; // Initialise each value separately.
    arr[0][1] = 5;
    arr[0][2] = -3;
    arr[1][0] = 0;
    arr[1][1] = 6;
    arr[1][2] = 7;

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

Declaring 2D Arrays

```cpp
// source/2d_arrays_002.cpp
#include <iostream>
using namespace std;

int main() {
    // Declare array at initialisation.
    int arr[2][3] = { { 2, 0, -3 }, { 4, 6, 7 } };

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

Declaring 2D Arrays

```cpp
// source/2d_arrays_003.cpp
#include <iostream>
using namespace std;

int main() {
    // Let the compiler make the splits.
    int arr[2][3] = { 2, 0, -3, 4, 6, 7 };

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

What Will This Print?

```cpp
// source/2d_arrays_004.cpp
#include <iostream>
using namespace std;

int main() {
    // Let the compiler make the splits.
    int arr[][] = { 2, 0, -3, 4, 6, 7 };

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

2D Array Dimension Declaration

Hopefully, you see something like the following in your console:

```
2d_arrays_004.cpp:7:9: error: declaration of ‘’arr as multidim
ensional array must have bounds for all dimensions except the
first
7 |     int arr[][] = { 2, 0, -3, 4, 6, 7 };
  |         ^~~
```

This is because, as the error says, when it comes to multidimensional arrays, **all but the first dimensions must be provided!**

How About This?

```cpp
// source/2d_arrays_005.cpp
#include <iostream>
using namespace std;

int main() {
    // Let the compiler make the splits.
    int arr[][3] = { 2, 0, -3, 4, 6, 7 };

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

2D Arrays And Functions

The same holds true when declaring arrays as function parameters:

```cpp
// source/2d_arrays_007.cpp
#include <iostream>
using namespace std;

int foo(int arr[][3], int rows, int cols) {
    return arr[rows - 1][cols - 1];
}

int main() {
    int arr[][3] = { 2, 0, -3, 4, 6, 7 };
    int x = foo(arr, 2, 3);
    cout << x << endl;
}
```

What Will This Print?

```cpp
// source/2d_arrays_006.cpp
#include <iostream>
using namespace std;

int main() {
    // Uncomment exactly 1 of the following 3 lines
    int arr[][3] = { 2, 0, -3, 4, 6, 7 };
    // int arr[][2] = { 2, 0, -3, 4, 6, 7 };
    // int arr[][6] = { 2, 0, -3, 4, 6, 7 };
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

2D Arrays Do Not Exist!

The previous code snippet was not expected to work, but it does for a single reason:

- 2D arrays **do not exist.**
- Indeed, what C++ does is to flatten the contents of a 2D array to consecutive memory locations.
- Thus, the `arr[i][j]` syntax does not actually mean “access the `arr` element at row `i` and column `j`”.
- But, then, how does C++ interpret `arr[i][j]`?

Rows And Columns

Using the following piece of code, can you figure what the C++ is doing behind the scenes when it comes to `arr[i][j]`?

```cpp
// source/2d_arrays_006.cpp
#include <iostream>
using namespace std;

int main() {
    // Uncomment exactly 1 of the following 3 lines
    int arr[][3] = { 2, 0, -3, 4, 6, 7 };
    // int arr[][2] = { 2, 0, -3, 4, 6, 7 };
    // int arr[][6] = { 2, 0, -3, 4, 6, 7 };
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

2D Array Flattening

C++ flattens arrays as follows:

- All elements are put in memory first according to their row and then based on their column.
- So, essentially, each element could be the element of a one–dimensional array at index k, as shown next.
- Compute k as a function of row number i and column number j?

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

2D Array Flattening

A flattened array's position index k is related to a 2D array's i, j indices by:

2D Array Flattening

A flattened array's position index k is related to a 2D array's i, j indices by:

$$k = i \cdot \text{\#columns} + j.$$

2D Array Flattening

A flattened array's position index k is related to a 2D array's i, j indices by:

$$k = i \cdot \text{\#columns} + j.$$

So, for instance, for a 2D array with 7 columns, the element at row 3 and column 4 (0–based indexing) should be placed at:

$$k = 3 \cdot 7 + 4 = 21 + 4 = 25,$$

i.e., at the 26th position.

2D Array Flattening

A flattened array's position index k is related to a 2D array's i, j indices by:

$$k = i \cdot \text{\#columns} + j.$$

So, for instance, for a 2D array with 7 columns, the element at row 3 and column 4 (0–based indexing) should be placed at:

$$k = 3 \cdot 7 + 4 = 21 + 4 = 25,$$

i.e., at the 26th position.
Can you see why we are allowed to omit (only) the first dimension in 2D array declaration?

2D Arrays Tips And Tricks

- 2D arrays are mostly used to make things conceptually easier for us (humans). They do not actually exist.
- So, use them whenever you need to make things easier to you.
- But, in general, this comes at a cost regarding memory de–allocation, as we shall see in the future, so be careful whenever you use 2D arrays!
- Alternatively, you can always use flattened 2D arrays, which should offload some memory management worries from you.
- Also, remember that you are allowed to not provide **only the first** dimension of a multidimensional array!

Strings Are Arrays

What will this print?

```cpp
// source/strings_001.cpp
#include <iostream>
using namespace std;

int main() {
    char msg[] = { 'H', 'i', '!', '\0' };
    cout << msg << endl;
}
```

Strings Are Arrays

What will this print?

```cpp
// source/strings_001.cpp
#include <iostream>
using namespace std;

int main() {
    char msg[] = { 'H', 'i', '!', '\0' };
    cout << msg << endl;
}
```

- This should print `Hi!`.

Strings Are Arrays

What will this print?

```cpp
// source/strings_001.cpp
#include <iostream>
using namespace std;

int main() {
    char msg[] = { 'H', 'i', '!', '\0' };
    cout << msg << endl;
}
```

- This should print `Hi!`.
- The `\0` at the end of the array is a special character, the `NULL` character which indicates the end of the string.

Using Double Quotes

Strings can also be initialised using double quotes, in which case the compiler adds the `NULL` character, so we do not need to insert it manually.

```cpp
// source/strings_002.cpp
#include <iostream>
using namespace std;

int main() {
    char msg[] = "Hi!";
    cout << msg << endl;
}
```

String Libraries

Since strings are arrays, we can manipulate them the same way we would with every other array, however we can also make use of the following libraries:

- `cctype`: character handling.
- `cstdio`: input / output.
- `cstdlib`: general utilities, some of them string-relevant.
- `cstring`: string manipulation.

String Cleanup

What will this print?

```cpp
// source/strings_003.cpp
#include <iostream>
#include <cctype>
using namespace std;

int main() {
    char str[] = "t6H0I9s6.iS.999a9.STRING";
    char c = str[0];
    for(int i = 0; c != '\0'; c = str[++i]) {
        if(isalpha(c))
            cout << (char)(isupper(c) ? tolower(c) : c);
        else if(ispunct(c))
            cout << ' ';
    }
    cout << endl;
}
```

String Operations

What will this print?

```cpp
// source/strings_004.cpp
#include <iostream>
#include <cstring>
using namespace std;

int main() {
  char fragment1[] = "I'm a s";
  char fragment2[] = "tring!";
  char fragment3[20];
  char finalString[20] = "";

  strcpy(fragment3, fragment1);
  strcat(finalString, fragment3);
  strcat(finalString, fragment2);
  cout << finalString << endl;
}
```

String Tips And Tricks

- When looping over strings, using the `NULL` character is a nice universal way to determine when all the string has been consumed. Thus, we need not pass string length as a parameter in string manipulation functions.
- Since each string contains the `NULL` character, all strings are by default non–empty!
- Remember that `chars` are declared using single quotes (’), while strings using double (”).
- Some string manipulation functions return strings while we would need a single `char`. In this case, we have to **cast the output** to a `char` before using it!

Fun Time!

In–class Exercise #001

Write a C++ program that:

- asks the user for a positive integer number, n;
- checks if the number is even or odd, and;
- prints on screen `even` if the number is even, `odd` otherwise.

In–class Exercise #002

A PC manufacturer has the following retail pricing catalogue:

- For orders with at most 100 PCs, unit cost is 560$ each.
- For orders with at most 250 PCs, the first 100 are priced as above and the rest at 480$ each.
- For orders with at most 400 PCs, the first 250 are priced as above and the rest at 400$ each.
- For orders above 400 PCs, the first 400 are priced as above and the rest at 320$ each.

Write a C++ program that accepts the number of PCs ordered by some customer and computes and prints on screen to total cost of that order.

In–class Exercise #003

Write a C++ program that:

- asks the user for a positive integer number, n;
- checks if the number is prime or not, and;
- prints on screen `prime` if the number is prime, `composite` otherwise.

As a reminder, a positive integer, n, is said to be prime if the following conditions hold (both of them):

- $n > 1$.
- The only divisors of n are 1 and n.

So, 2, 7, 13 and 19 are some primes while 4, 15 and 21 are not.

In–class Exercise #004

Consider the C++ code shown right.

- Without compiling and executing it, what do you expect it to do?
- Compile and run that program. What did it print?
- Can you explain it?

```cpp
// source/exercise004.cpp
#include <iostream>
using namespace std;

int main() {
    double x = 1.0;
    int i = 0;
    while (x > 0) {
        x = x / 2;
        i++;
    }
    cout << x << ", " << i << "\n";
    return 0;
}
```

In–class Exercise #005

Consider the C++ code shown right.

- Without compiling and executing it, what do you expect it to do?
- Compile and run that program. What did it print?
- Can you explain it?

```cpp
// source/exercise005.cpp
#include <iostream>
using namespace std;

int main() {
    double x = 1.0;
    while (x / 2 > 0) {
        x = x / 2;
    }
    double y = 1.4 * x;
    bool b = x == y;
    cout << b << "\n";
    return 0;
}
```

In–class Exercise #006: Advanced Pointer Fun

While we have said enough about pointers, we have not explored pointer–land in full. The following tutorial will help you do so:

`https://learnmoderncpp.com/arrays-pointers-and-loops/`

Follow the tutorial step–by–step and pay attention to the “Experiments” it asks you to execute. Write down your observations in a document, which you will share with me at the end of the class at: `v.markos@mc-class.gr`.

In–class Exercise #007

Implement the following functions in C++:

1. A function, `add()`, that takes as arguments two 3×3 `double` arrays and returns their sum.
2. A function, `transpose()`, that takes as argument a single 3×3 `double` array, `arr`, and returns its transpose, i.e., a 3×3 array whose rows are the columns of `arr`.
3. A function, `diag()`, that takes a 3×3 double array and computes and return the sum of its diagonal elements.

In–class Exercise #008

Implement a C++ function that:

- takes a one dimensional `int` array of length 25, and;
- prints the array's elements in a spiral order, i.e., starts from top left and, moving first right, then down, then left and then up, prints elements spiral–wise.

In–class Exercise #009

Implement tic–tac–toe in C++ as follows:

- Create a 3×3 matrix to represent the game board.
- Implement functions to make moves, check for wins, and check for draws.
- Play the game interactively.

In–class Exercise #010

The dot product of two vectors is computed as follows:

$$(x_1, x_2, x_3) \cdot (y_1, y_2, y_3) = x_1 y_1 + x_2 y_2 + x_3 y_3.$$

Write a C++ function that takes two 3 dimensional vectors as arguments and computes and returns their dot product.

In–class Exercise #011

We can multiply two square 2×2 matrices as shown below:

$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}\begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} = \begin{pmatrix} a_1b_1 + a_2b_3 & a_1b_2 + a_2b_4 \\ a_3b_1 + a_4b_3 & a_3b_2 + a_4b_4 \end{pmatrix}.$$

Write a C++ function that takes two 2×2 double arrays and computes their product.

In–class Exercise #012

The determinant of a 2×2 matrix is given by the following formula:

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc.$$

Write a C++ function that computes the determinant of a 2×2 double array.

In–class Exercise #013

The determinant of a 3×3 matrix is given by the following formula:

$$\det \begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{pmatrix} = a_1 \det \begin{pmatrix} a_5 & a_6 \\ a_8 & a_9 \end{pmatrix} - a_2 \det \begin{pmatrix} a_1 & a_3 \\ a_7 & a_9 \end{pmatrix} + a_3 \det \begin{pmatrix} a_4 & a_5 \\ a_7 & a_8 \end{pmatrix}.$$

Write a C++ function that computes the determinant of a 3×3 double array.

In–class Exercise #014: Part A

This is a three part self–study exercise. At first, read the following Wikipedia paragraph about how PPM image files are structured:

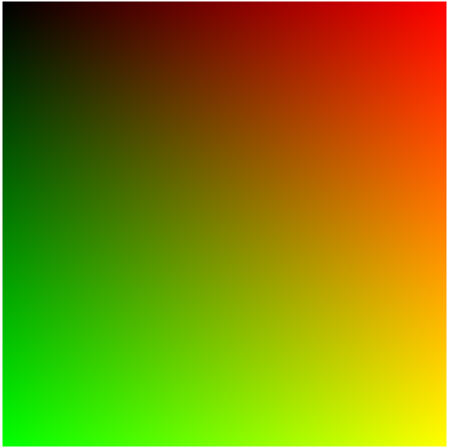
`https://en.wikipedia.org/wiki/Netpbm#PPM_example`

Then implement a python function that creates a 256×256 `.ppm` red image file.

Regarding C++ and file handling, either recall your C knowledge or look around the web!

In–class Exercise #014: Part B

As your first actual exercise with PPM images, try to generate an image like the one shown right. To do so, you might find useful to recall how red and green are represented in RGB.



In–class Exercise #014: Part C

Being sufficiently exposed to PPM images and C++, you now have to implement the following C++ functions:

- a function, `flipX()` that flips an image across the horizontal axis;
- a function, `flipY()` that flips an image across the vertical axis;
- a function, `grayscale()` that turns an image into grayscale.

You can use the image generated in part B to test your functions.

In–class Exercise #015

Create the following functions in C++:

1. An int function, `substrSearch()` that takes two strings, `needle` and `haystack` and looks for the first occurrence of `needle` in `haystack` and returns its starting index.
2. A boolean function `isPalindrome()` that takes a string and checks whether it is a palindrome, i.e., whether it reads the same right–to–left and left–to–right.
3. A boolean function `isAnagram()` that takes two strings, `ana` and `gram` and checks if `ana` is an anagram of `gram`, ignoring case and spaces.

In–class Exercise #016

Implement a C++ function `strComp()` that:

- takes as input a string, `str`;
- checks that it contains only characters in the range `a-z` or `A-Z`;
- spots any characters that repeat at consecutive positions, and;
- returns a string with consecutively occurring characters replaced by a single instance of that character followed by an integer indicating the number of repetitions.

For instance, for input `aaaabcccd` it should return `a4bc3d`.

In–class Exercise #017

Read about the Knuth–Morris–Pratt substring search algorithm:

`https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm`

Then implement it in C++ with an appropriate function and any other required machinery.

Homework

Complete all exercises and problems in MIT's C++ course first assignment, found here:

`https://ocw.mit.edu/courses/6-096-introduction-to-c-january-iap-2011/resources/mit6_096iap11_assn01/`

For your convenience, you can also find the assignment file in this lecture's materials, at: `../homework/MIT6-096IAP11-assn01.pdf`. Submit all your work in the online form below as a single `.zip` file:

`https://forms.gle/rSq3VSpcouRAVjqMA`

or via email at: `v.markos@mc-class.gr`.

Homework

Complete all exercises and problems in MIT's C++ course second assignment, found here:

https://ocw.mit.edu/courses/6-096-introduction-to-c-january-iap-2011/797ebff419fa2cc3a10af2c5f19be961_MIT6_096IAP11_assn02.pdf

For your convenience, you can also find the assignment file in this lecture's materials, at: ../homework/MIT6-096IAP11-assn02.pdf. Submit all your work in the online form below as a single .zip file:

https://forms.gle/rSq3VSpcouRAVjqMA

or via email at: v.markos@mc-class.gr.

Homework

1. Complete any in–class exercises you haven't so far.
2. Since this course's aim is to study socket programming in C++, this homework is mostly oriented towards that direction, provided we have studied enough C++ so far. To get yourselves comfortable with sockets in C++, study the tutorial found below:

   `https://beej.us/guide/bgnet/html/`

Share your comments and implementations at: `v.markos@mc-class.gr`

Any Questions?

Do not forget to fill in the questionnaire shown right!

https://forms.gle/dKSrmE1VRVWqxBGZA