TDT4195: Visual Computing Fundamentals

Assignment 0: Introduction into C++

August 20, 2019

Bart van Blokland
Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)

- This assignment is **OPTIONAL**. It does not count towards your final grade. However, it is highly recommended you work through it.

Overview

This optional assignment aims to introduce you to the parts of C++ you need to complete the assignments in this course. It is by no means a complete introduction, nor is it entirely accurate.

Due to its C heritage, and complexity in the hardware the language tries to abstract, parts of it can appear illogical, inconsistent, and complicated.

I will therefore present the language by abstracting a lot of details away, hopefully allowing you to make sense and use the language effectively for the assignments in this course.

I will also assume you are familiar with Java; a language that shares much of C's syntax.

Compiling a first program

To start things off, we need to make sure your system has all necessary tools installed to compile C++ programs. The lab machines already have these installed, if you want to work on those. You will at least need the following installed:

- CMake
- g++
- git

For Windows, instead of g++, make sure you have visual studio, or any other compiler installed.

If you're running Ubuntu, you can install all of these using the following command in the terminal:

```
sudo apt install cmake g++ git
```

In order to take away some of the pain in setting up a new C++ project, I've created a sample one for you to use. Navigate to a folder where you'd like your code to live, and use git to download a copy:

```
git clone https://github.com/bartvbl/TDT4200-Assignment-0
```

Next, navigate into the *build* directory inside the project, and run CMake:

```
cd TDT4200-Assignment-0/build
cmake ..
```

You will need to run the `cmake` command every single time after you add a new source file to your project. It will automatically add the new source file to the list of files being compiled in the Makefile.

To build the project on Linux, run `make`.

To build the project on Windows, open up the .sln (solution) file generated by cmake in visual studio, and select the "build solution" option in the "build" menu.

The output from `make` should inform you the project was built successfully. You can try running the executable using:

```
cintro/cintro
```

Notice that you will need to run this from the "build" directory. The program should print a single line of output.

If it did, we can move on to looking at some syntax.

Variables

Variable declaration is pretty much entirely the same as in java, and data types such as `int`, `float`, and `double`, also exist in C. There is, however, one important difference I should mention.

If I declare two integers:

```
int foo = 5;
int bar;
```

The variable `foo` will contain the value of 5. In Java, `bar` would have the value of 0 after executing the above snippet, which is the default value of an integer when no value is specified. In C, however, variables that have not been assigned a value do not take on a default value. Instead, their initial value is whatever value the bit of memory assigned to the variable happens to contain.

To avoid confusion, I recommend always initialising your variables with a default value (assigning a value to any variable you define). It can save you a lot of debugging hours searching for a problem that only sometimes occurs.

It is also possible to declare constants. The values of these must be known at compile time. Declaring a constant can be done through the `const` keyword:

```
    const float pi = 3.1415926;
```

Unlike Java, C++'s boolean datatype is `bool`.

Casting (changing a variable's type) works exactly the same as in Java:

```
float d = 3.7;
int intValue = (int) d; // intValue = 3
```

Functions

A function in C++ is defined just like it is in Java, except functions are more standalone (they are not defined in a class). Their syntax is mostly the same:

```
return_type functionName(type1 argument1, type2 argument2) {
    return someValue;
}
```

Here's a more concrete example:

```
int addTwoIntegers(int a, int b) {
    return a + b;
}
```

Writing to the terminal/command prompt

Writing text and values to the command line through the "standard out" stream is a useful way to debug your program and to get a feel for what is going on. Using it is fairly straightforward.

First, you need to import the standard IO streams, by writing the following line near the top of your source file:

```
#include <iostream>
```

Next, you can print a range of basic data types such as `std::string`[1], `int`, `float`, and `bool`, using:

```
// std::endl represents your system's newline character(s)
std::cout << "This is a string" << std::endl;

// You can print multiple pieces of data at once:
std::cout << "String 1 " << "String 2" << std::endl;

// Which do not need to be of the same type:
std::cout << "The value five: " << 5 << std::endl;

// And can be stored in variables:
int eight = 9;
std::cout << "The value after 7 is: " << eight << std::endl;
```

You'll find more examples in the remainder of this document.

[1]Due to C++'s C heritage, strings are handled the way C does by default. Std::string is the closest equivalent to Java's String.

Defining functions in other files

For starters, functions within a file can only be used if they have been declared above a piece of code. So for two functions defined like this:

```
void functionOne() {
}
void functionTwo() {
}
```

It is possible to call `functionOne()` from `functionTwo()`, but not vice versa. However, this *is* possible by *declaring* the function first, and *defining* it later. Here's how:

```
// The function is declared here
void someFunction(int argument1);

// And defined here
// Note their 'signatures' (name, parameters, and return type)
// are equivalent.
void someFunction(int argument1) {
    doSomething();
}
```

Because the above function is declared prior to it being defined, it is possible to use it if you were to define a new function between the declaration and definition in the above snippet. The declaration tells the compiler a function with that signature will be defined somewhere in the future as it works its way from the top to the bottom of the file, thereby allowing it to be used in other functions.

C/C++ source files are almost always accompanied by a so-called "header file". The idea is that functions are declared in the header file, and defined in the source file [2].

The process of creating a new source file is as follows:

1. Create two empty files. Give them the same name, but extensions ".hpp" and ".cpp", respectively.

[2] The reason for why this system exists is that in the days C was developed, data was primarily stored on tape drives, which were incredibly slow. By putting all function declarations in a separate file, the compiler was able to compile source code significantly faster. C++ has inherited this system. There is absolutely no reason for their existence nowadays. An *#import* statement similar to languages such as Java or Python is in the works, but it won't be available in the language for some years to come. Until that time we're unfortunately stuck with the old junk that are header files.

2. Open the ".cpp" file. Write `#include "yourheaderfile.hpp"` at the top (replace the filename with the one of your header file). You need to specify a path here if your header file is stored in a subdirectory in your project.

3. Open the ".hpp" file. Write `#pragma once` at the top.

4. Re-run CMake in your build directory, so that the new source file is added to the build process.

You can now put function *declarations* in your header file, and their *definitions* in the .cpp source file. If you need to use functions from your new source file from another one, all you need to do is use an `#include` directive in the source file where you want to use your functions to include the *header* file you just created. This works similarly to using the `import` statement in Java[3].

[3] I'm basically omitting the entire C++ build process here. In reality, #include statements *copy and paste* the contents of one file into another, and a separate "linker" compilation stage tries to figure out which function is located in which file after compiling each individual source file separately. It's another design decision that made a lot of sense around the time C was designed, but nowadays has become a giant bottleneck instead. If modern languages seem to compile significantly faster than C++, this is one of the reasons why.

Pointers

Pointers are variables whose value contains the memory address of another value. For this course that's really everything you need to know about them.

You can think of pointers as something similar to locations on a map. The value of a pointer is in effect a location, and when you have that location, you can go see what is there. The key thing here is that while a location of something doesn't change, what you find there *can*.

There are several reasons why using a *reference* to a value can be beneficial. For instance, if a variable is shared between a number of objects, you can have all objects store a pointer instead. Updating the value which the pointer points to updates the value for all objects simultaneously. Additionally, passing a reference to a variable in a function allows the function perform modifications on that variable without you explicitly having to use the return value for that.

To obtain the memory address of a variable, you can use the `&` operator, and to "dereference" a pointer (request the value located at a pointer's address), you can use the `*` operator. Here's an example:

```
int foo = 5;

// This creates a pointer to the variable foo.
// Notice that the pointer has the same type as the variable,
// but has an * behind it to indicate it is a pointer.
int* pointerToFoo = &foo;

// It is also possible to create a pointer to a pointer:
int** pointerToPointer = &pointerToFoo;

// Dereferencing a pointer is done using the * operator:
int bar = *(pointerToFoo); // bar = 5

// Dereferencing a pointer to a pointer.
// The first dereference operator obtains the address of the pointer to foo.
// The second one yields the value of foo, which is assigned to door.
int door = *(*(pointerToPointer)); // door = 5

// Here's how to update the value at the address stored in a pointer:
*(pointerToFoo) = 10; // sets foo to 10
```

Pointers are useful in a number of situations. In this course you'll mostly need to use them for memory management and arrays. However, they can also be useful when you'd like to share a large data structure without having to make copies every time you need to share it with another function.

On the other hand, they can be difficult to manage and a frequent source of errors. It is better to avoid them as much as possible, for instance by using a container type such as `std::vector` (which I'll show you later).

In particular, if you for example try to read a value past the end of an array, your program may crash due to a so-called "segmentation fault". These errors occur when the operating system detects that your program is trying to read from an area of memory that has not been allocated to it. This causes the operating system to immediately terminate your program.

Pointer variables can be initialised with the special **`nullptr`** value. This value is a memory address which is guaranteed to point to an invalid address. Dereferencing it triggers a segmentation fault.

Memory

Understanding memory management in C++ is the primary thing I hope you take away from this assignment / tutorial.

In C++, there are two primary areas of memory; the stack and the heap [4]. The stack is an area in memory with a limited size where the program can store the values of small temporary variables. The way the program uses it (as we'll see in a bit) is just like the the name suggests: it pushes values on it to allocate them, and pops them off when it is done using them. The heap allows you to allocate comparatively large amounts of memory for storing bulkier or long-term data.

For understanding the stack in detail, we first need to understand the concept of a "scope". A scope is anything contained within a pair of braces:

```
// This is the start of one scope
{
    // Here I define a scope within a scope
    {

    } // <- Inner scope ends here
} // <- Outer scope ends here
```

You can nest as many scopes within each other as you'd like. Also note that statements such as `if`, `for`, `while`, as well as functions contain scopes too:

```
void someFunction()
{ // <- Here starts a scope
    if(someCondition)
    { // <- This is also a scope
        while(true)
        { // <- A deeply nested scope
            { // <- Another scope just because I can

            }
        }
    }
}
```

[4]The C++ standard never mentions the stack and heap explicitly. However, the language is definitely designed to accommodate them.

Variables defined within a scope are always allocated on the stack. The variables contained within a scope are all allocated at once at the *start* of that scope, and deleted when it ends. For instance, here's what the stack looks like on a line-by-line basis for a small program:

```
// Stack: <empty>
{
// Stack: int a, int b, float c
    int a;
// Stack: int a, int b, float c
    int b;
// Stack: int a, int b, float c
    float c;
// Stack: int a, int b, float c
    for(int i = 0, i < 10; i++) {
// Stack: int a, int b, float c, int i, float x, float y
        float x;
// Stack: int a, int b, float c, int i, float x, float y
        float y;
// Stack: int a, int b, float c, int i, float x, float y
        if(x > y) {
// Stack: int a, int b, float c, int i, float x, float y, float k
            float k = x + y;
        }
// Stack: int a, int b, float c, int i, float x, float y
        if(x < y) {
// Stack: int a, int b, float c, int i, float x, float y, float m
            float m = x - y;
        }
// Stack: int a, int b, float c, int i, float x, float y
    }
// Stack: int a, int b, float c
}
// Stack: <empty>
```

Note that the variables `k` and `m` occupy the same location in memory. Once the scope containing `k` exits, the same location can be reused for the value of `m` when that stack frame is pushed on to the stack.

The memory layout itself is something you don't need to be concerned about, apart from understanding that stack variables are allocated at the start of a scope, and deallocated at the end of one. The compiler handles memory layout for you.

When you call a function, the function's scope is pushed on to the stack first, and subsequently any scopes defined within that function, just as if it had been another scope within the function calling it. The same is true for any recursive or nested calls from that function.

The other main memory type is that which is allocated on the heap. As I mentioned previously, the stack has a limited size [5], which is normally more than sufficient to store all temporary variables you might need. The only way to cause it to run out quickly is to either allocate large chunks of memory on the stack, or do some extremely deep recursion.

For cases where you *do* need to store large quantities of memory, you can allocate it on the heap. You can allocate any data type you desire, by using the `new` operator:

```
int* singleIntegerOnHeap = new int;
float* singleFloatOnHeap = new float;

// Structs can also be allocated on the heap
Point* point = new Point;
```

Note that the new operator returns a *pointer* to the allocated memory.

Heap allocated memory has the nice property that it is not deallocated when a scope exits, as is the case with its stack-allocated counterpart. However, you become responsible for deallocating it when you're done yourself. You can delete heap-allocated memory using the aptly named `delete` operator:

```
delete point;
```

Whenever you allocate memory on the heap, you should also immediately add a deallocation statement when you're done with it. If the pointer variable is deleted when the scope ends, you lose the reference to the heap-allocated memory permanently. You're at that point no longer able to free it. This problem is known as a "memory leak".

There are a couple of other things to watch out for. First, using heap-allocated memory after deleting it. This means your heap pointer points to memory that's no longer part of your process, so dereferencing that pointer causes an error. This situation is known as a "dangling pointer".

Second, heap-allocated memory should be allocated once, and freed exactly once.

[5] The actual size can be configured, but the defaults vary between compilers. As a rule of thumb, Windows uses a default of 1MB, OSX and Linux reserve 8MB.

Arrays and Vectors

To allocate an array, you can use the `new[]` operator:

```
int* integerArray = new int[10];

// Size can also be defined by a variable
int desiredLength = 9001;
int* bigArray = new int[desiredLength];
```

Note that because we're using the **`new`** operator here, the array is allocated on the heap. The pointer you receive points to the first element of the array.

Deleting an array (deallocating it) requires a special bit of syntax. Any arrays allocated with the **`new`**`[]` operator *must* be deallocated using the **`delete`**`[]` operator. Here's an example:

```
int* spam = new int[42];
delete[] spam;
```

You can set the value of an element just as you would in Java:

```
integerArray[4] = 2;
```

Reading values works the same too:

```
int abc = integerArray[3];
```

The major downside with arrays in C++ is that they are stored as a pointer to the first element of the array. This works fine for reading and writing to their contents, *given that you know their length*. In Java, requesting element 11 from an array of 10 elements will throw an `IndexOutOfBoundsException`. In C++, at best the program will crash with a segmentation fault, but at worst will read a completely unrelated piece of memory and happily carry on.

You will therefore need to make sure you keep track of the array length in a separate variable, and check for whether you're reading out of bounds indices manually.

Additionally, arrays have a constant size. Once they are allocated, extending it by one element requires you to allocate a new one and copy over all of its contents.

Fortunately, C++ also has a container type which *does* include bounds checking, and is able to manage its own memory; `std::vector`. The `std::vector` is a data structure much like Java's `ArrayList`. Here's how to use it:

```cpp
// Creating a new vector containing integers
// You can replace the int datatype with any other type you desire
std::vector<int> someVector;

// Appending an item
someVector.push_back(5);

// Setting an element at index 4
someVector.at(4) = 2;

// Requesting the value of the element at index 6
// Notice we both use the at() function for reading and writing.
std::cout << "Value of index 6 is: " << someVector.at(6) << std::endl;

// Remove the 5th element in the vector
someVector.erase(someVector.begin() + 4);
```

Before you can use vectors, you need to "import" them by writing `#include <vector>` near the top of your source file.

Vectors also allow you to read their contents using the `[]` operator, just like you use to read the contents of arrays. For example:

```cpp
int indexed = someVector[4];
// Is equivalent to
int indexed = someVector.at(4);
```

There is, however, one important difference. The `[]` operator does not check for array bounds, while the `.at()` function does. I therefore recommend you always use the `.at()` function.

Vectors manage their own memory internally, whose contents are automatically stored on the heap (though the metadata structure is usually allocated on the stack, as is the case in the snippet above). If you know the size of a vector in advance, you can hint at it to preallocate the amount of memory you need. For large vectors this means the memory manager needs to be invoked far less often. You can use the `reserve()` method to ensure the vector allocates the space you need in one go:

```
someVector.reserve(1000);
```

Vectors hang on to the memory they've reserved even when you clear their contents. If you want to reuse a vector, but reduce the amount of memory it uses, you can use the `shrink_to_fit()` function, which has the opposite effect of `reserve()`.

In order to quickly change the *length* of your vector to a certain number of elements, you can use the `resize()` function:

```
std::vector<int> growingVector;
std::cout << growingVector.size() << std::endl; // Prints 0

growingVector.reserve(1000);
std::cout << growingVector.size() << std::endl; // Prints 0

growingVector.resize(1000);
std::cout << growingVector.size() << std::endl; // Prints 1000
```

Structs

Data structures, or "struct" for short, are a mechanism to group variables together. This is useful when data logically fits together, such as with a 2D location:

```
typedef struct Point {
    float x;
    float y;
} Point;
```

Structs (as well as classes; they're mostly the same in C++) require you to use a special bit of syntax when you're requesting the values of a struct field through a pointer. Here's how to do it, using the `Point` struct I declared above:

```
Point point;
point.x = 5;
point.y = 8;

Point* pointerToPoint = &point;

// The normal way to request the value of a struct field:
std::cout << "normal: " << point.x << std::endl;
// Through a pointer instead requires using the -> operator:
std::cout << "pointer: " << pointerToPoint->x << std::endl;
```

Structs are laid out consecutively in memory when allocated as an array. So an array of 4 instances of Point (as defined above) looks like this:

```
float x, float y, float x, float y, float x, float y, float x, float y
```

It's not something you need to be concerned about when writing code, however, it is useful to know about for optimising data locality (as we will see in the lectures).

Going for full performance

By default, CMake creates a makefile which does not apply compiler optimisations. These, as the name implies, make your program run a lot faster. Meanwhile, they also remove many safety checks and make the code much harder to debug. So you should only turn them on once you know your program is (as good as) free of bugs.

To turn them on, you need to run CMake with a special command line argument:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

You can subsequently run `make` to rebuild your program.

And that's all there's to it! If you want to disable compiler optimisations, just run CMake again without the `CMAKE_BUILD_TYPE` argument.

Task 1: Questions

a) Compile and run the sample project, as shown on page 3.

b) Consider the following snippet of code:

```cpp
#include <iostream>

typedef struct Point {
        float x;
        float y;
        float z;
} Point;

Point* createPoint() {
        Point point;
        point.x = 3;
        point.y = 1;
        point.z = 4;
        return &point;
}

int main(int argc, char** argv) {
        Point* point = createPoint();
        std::cout
        << point->x << ", "
        << point->y << ", "
        << point->z << std::endl;

        return 0;
}
```

Executing this code results in so-called “undefined behaviour”, due to a problem related to memory management (although in most C++ implementations it will run correctly regardless).

What is the problem?

What is one way to fix it?

c) Here's another snippet of code:

```
int compute(std::vector<int> list1, std::vector<int> list2) {
    int* elementProducts = new int[list1.size()];

    for(int i = 0; i < list1.size(); i++) {
        elementProducts[i] = list1.at(i) * list2.at(i);
    }

    int totalValue = 0;
    for(int i = 0; i < list1.size() - 1; i++) {
        totalValue += list1.at(i) - list1.at(i + 1);
    }

    return totalValue;
}
```

This snippet contains a memory management problem too.

You may assume list1 and list2 always are the same length.

What is the problem?

What is one way to fix it?

d) Here's yet another snippet:

```
#include <iostream>
#include <vector>

int main(int argc, char **argv) {
        std::vector<int> integerList;
        integerList.resize(1000);

        for(int i = 0; i < 1000; i++) {
                integerList.at(i) = i;
        }

        int a;
        int b;

        for(int i = 1; i < 1000; i++) {
                b = integerList.at(i);
                integerList.at(i - 1) = a + b;
                a = b;
        }

        int sum = 0;

        for(int i = 0; i < 1000; i++) {
                sum += integerList.at(i);
        }

        std::cout << sum << std::endl;

        return 0;
}
```

When running this program multiple times consecutively, the value it prints to the command line can vary between runs.

What is the cause of this?

e) And yet another snippet with a memory related issue:

```
#include <iostream>
#include <vector>

int main(int argc, char **argv) {
        std::vector<int> integerList;
        integerList.resize(1000);

        for(int i = 0; i < 1000; i++) {
                integerList.at(i) = i;
        }

        // Using a value that at least has 64 bits
        unsigned long long sum = 0;

        int* subArray;
        for(int i = 0; i < 1000; i++) {

                int subArrayLength = 1000 - i;
                subArray = new int[subArrayLength];

                for(int j = i; j < 1000; j++) {
                        subArray[j - i] = i * j;
                }

                for(int k = 0; k < subArrayLength; k++) {
                        sum += subArray[k];
                }
        }
        delete[] subArray;

        std::cout << sum << std::endl;

        return 0;
}
```

What memory-related problem exists within this snippet?

What is one way of resolving it?

f) Let’s play some “Cups and balls”, with pointers!

Try and find its output without compiling it, if you dare...

```
#include <iostream>

int main(int argc, char **argv) {
        int a = 1;
        int b = 2;
        int c = 3;

        int* p1 = &a;
        c = a;
        int* p2 = &(*(p1));
        int* p3 = &c;
        b++;
        int* p4 = *(&p3);
        int** p5 = &p1;
        p4 = p2;
        *p5 = p1;
        int** p6 = &p4;
        (*(p3))++;
        *p4 -= 2;
        *p5 = p2;
        *p1--;
        *p6 = p3;
        a++;
        std::cout << *(*(p6)) << std::endl;
}
```