**Compilers and Optimization**

As mentioned in the video, a C++ compiler rewrites your code into binary instructions. Many compilers will try to optimize code for you. These optimizations aim to

* make the code run faster,
* use less memory, or
* consume less electric power.

Compilers can be quite good at optimizing as they write your C++ code into machine code; however, when running the same code through different compilers, you might find that the slowest parts of your code are not the same in each case. Hence, it's important to find the weak spots by actually measuring how long it takes to execute or how many resources the code uses. You'll learn more about measuring your code later in the lesson.

**Hardware Limitations**

Hardware can put limitations on your programs and lead to code inefficiencies.

In embedded systems engineering, you might not always be working with the newest and most powerful computer processors. A typical example would be trigonometric functions. Some computer architectures might use relatively slow software approximations for a sine function rather than using the arithmetic/logic unit. If you can use an approximation that runs directly on the CPU's arithmetic/logic unit, you might get your code to run faster.

A screenshot of a cell phone

Description automatically generated

## C++ Demonstrations

This lesson contains a series of demonstrations showing how C++ manages memory. You'll see in the next part that these demonstrations are embedded into the classroom.

Each demonstration is contained in a separate folder with a set of instructions and code. The next section gives more details about how you will run the demonstrations.

### **Assembly Language**

In between C++ and machine code there is actually another language called assembly language. Assembly language is a human readable low-level language that gets you even closer to the hardware than C++.

Your compiler might not actually produce assembly language code and instead go directly to machine code. But you can still see assembly language code if you're curious. And there might be rare cases when you are trying to improve code efficiency and write assembly language directly in order to improve performance.

In the previous demonstration, you can use the following commands to output the assembly language code:

cd ~/home/workspace/demo\_machine\_code

g++ -S machine\_code.cpp

This will output a file called machine\_code.s, which you can then double-click to see its contents.

Assembly language is not nearly as intuitive as C++, but it is still human readable. When you do something as simple as declaring and defining a variable,int x = 5;. the computer has to break this up into a series of steps like assigning a space in memory to the variable x and then placing the value 5 into the assigned space.

You can write out each of these steps directly in assembly language. For the purposes of this course, you do not need to be familiar with assembly language. But just looking at assembly code will prove to you that every line of C++ code has consequences in terms of efficiency. Look at the number of steps involved with just assigning the value to x.

Unnecessary lines of code mean that the CPU will take more time to execute a program than what is actually needed.

## How much space do my variables use?

The [**standard that defines C++**](https://isocpp.org/std/the-standard) specifies the minimum number of bytes required for each variable type. For instance, an integer is guaranteed to have at least 2 bytes or 16 bits.

That does not mean your computer will use 16 bit integers by default. The default number of bits will depend on how your computer system was designed. An int variable might be 16 bits on some systems but 32 bits on other systems. You can get more information at this [**link**](http://en.cppreference.com/w/cpp/language/types).

Although the exercises later in this lesson will focus on increasing the speed of your code, you might find yourself at other times trying to optimize for memory use as well. The more comfortable you are with how your computer works, the more tools you will have for optimization.

## All Variables are Binary

C++ stores all variables in multiples of bytes:

* a char is 1 byte
* a 16-bit integer would be 2 bytes
* a 32-bit integer would be 4 bytes
* a 32-bit float would be 4 bytes

Thus all variables are represented by binary number. A 32-bit integer and a 32-bit float number take up the same amount of space; they would both be represented by a series of thirty-two 0s and 1s.

## How Bytes Limit Value Ranges

In C++, your variables will take up either 8, 16, 32, or 64 bits of memory, which are 1, 2, 4, and 8 bytes respectively.

The number of bytes put a limit on the minimum and maximum values that your variables can hold. As mentioned in the video, a 32 bit integer can have a maximum value of 4,294,967,295; however, if a variable might take on either a positive or negative value, then you need to use one bit to represent the variable's sign. This leaves 31 bits to represent the integer giving a max value of 2,147,483,647.

Likewise, 32-bit floats can only contain about seven decimal places whereas 64-bit doubles can have about 15. The explanation for how floats are stored is a bit complex; however, you can imagine that a fixed number of bits puts a limit on the amount of decimal places that can be kept. In fact, you'll see this in an upcoming demo.

## Is my system 32 bit or 64 bit?

Your CPU probably either has a 32-bit architecture or a 64-bit architecture. That means the CPU was designed to work well with storing and manipulating information in 32-bit chunks versus 64 bit chunks.

If your CPU uses a 32-bit architecture, you can still create 64-bit variables in your programs as long as your compiler has this feature. But, the code will most likely run more slowly than using a 64-bit architecture with 64-bit variables. On a 32 bit system, the compiler has to create extra instructions to move and do math on 64-bit variables.

If you'd like to see whether your computer has a 32-bit or 64-bit system, here are instructions for:

* [**windows**](https://support.microsoft.com/en-us/help/827218/how-to-determine-whether-a-computer-is-running-a-32-bit-version-or-64)
* [**mac**](https://apple.stackexchange.com/questions/12666/how-to-check-whether-my-intel-based-mac-is-32-bit-or-64-bit)

## Stack vs Heap

##### Stack

When you declare a variable in C++, the variable will automatically be placed on the stack. Once a function terminates, for example the main function, then the variable is removed from the stack; however, in terms of the code you've written so far in the nanodegree, there is one exception. The elements in a vector actually get placed on the heap, but the compiler still manages the allocation and deallocation of memory for you.

The stack removes variables by the "last in first out" rule; in other words, the last variable to be placed on the stack will be the first variable removed from the stack. This makes sense given that when a function is called, variables will be allocated to memory and then when the function terminates, variables will be removed from memory.

The stack also tends to be relatively small: perhaps 1 MB depending on your system. One advantage to keeping the stack small is for multi-threading. Let's say you only have 50 MB of RAM for the stack. Your CPU could do about 50 simultaneous tasks because of the smaller stack size. But because the stack is small, the stack can run out of memory; this is called stack overflow.

#### Heap

The heap, on the other hand, is only limited by the amount of RAM currently available. So variables that hold a lot of memory have to go on the heap. But when you declare a variable on the heap, you are responsible for removing the variable from memory. If you don't, then it becomes more likely that your program will run out of memory before the program terminates. And then your program will crash.

The heap also tends to be slower; a compiler organizes the stack for you and knows where the next available memory slot is; on the other hand, a program might have to search for an empty spot to put a variable on the heap.

In relation to code efficiency, only use the heap when necessary. Although you will not need to use the heap in the nanodegree, you'll at least become familiar with the syntax so that you can recognize when a program is using the heap. In the next section, you'll also see a demonstration about the stack versus the heap.

## Variables and Memory

Variables make programming much easier. Imagine what programming would be like if variables did not exist; you would have to determine

* determine how many bytes your variable needs
* find an available address to store the value
* make sure there are enough consecutive bytes available for storage
* you would also have to remember what value was stored at each hexadecimal address so that you could retrieve the right value as needed.

But with variables, the compiler does all of the memory management for you. And you can use descriptive names to help you remember what is contained in each variable.

This is essentially what the compiler is doing for you in terms of variables and memory management; without you having to think about it, the compiler efficiently finds space for your variables and keeps track of their location.

## Dynamic Memory

To understand the next demo, you need to know about dynamic memory allocation and pointers.

Dynamic memory allocation refers to when you, the programmer, assign variables to memory manually. These variables will go on the heap rather than the stack.

The opposite of dynamic memory allocation would be static memory allocation. You've already been using static memory in your programs; when you declare variables in your programs, the compiler knows ahead of time how much memory each variable will need; the amount of memory your variables need does not change as the program executes, so this memory is "static". The stack is used for static memory allocation.

The compiler doesn't know how much memory will be needed for dynamically allocated variables; hence, dynamic memory gets allocated when you execute your programs. Dynamically allocated variables go on the heap.

To use dynamic memory, you need to be familiar with **pointers** and the **new** and **delete** C++ syntax. A pointer is a special type of variable that holds a memory address rather than a value. You don't need to know how to use pointers, but they show up in the demo in the next part of the lesson.

Here is an example of dynamic memory allocation using pointers:

**#include <iostream>**

**int** **main**() {

*// asterisk syntax creates a pointer variable, which can hold a memory address*

**int** \* pointervariable;

*// new is used to create a variable on the heap. This line*

*// assigns an addresss to pointervariable and reserves enough space*

*// told hold an integer.*

pointervariable = **new** **int**;

*// Pointer variable holds an addresss. The address allows placing a value in*

*// memory at the address.*

\*pointervariable = 10;

std::cout << "pointer value: " << \*pointervariable << "\n";

std::cout << "pointer address: " << pointervariable << "\n";

*// remove pointervariable from the heap*

**delete** pointervariable;

pointervariable = NULL;

**return** 0;

}

With the result outputting something like:

pointer value: 10

pointer address: 0x1004053c0

although the exact memory address will differ from machine to machine. The pointer address is a [**hexadecimal number**](https://en.wikipedia.org/wiki/Hexadecimal) representing the location in memory.

The new operator assigns memory to the heap. You are are responsible for removing the variable when you are done with it, which is what the delete operator is for. Setting the pointer to NULL is good practice.

If you do not remove the variable, your program could run out of memory during execution; some operating systems might delete memory from the heap when your program terminates but some might not. Forgetting to remove dynamically allocated variables is called a **memory leak**.

Newer versions of C++ also include [**smart pointers**](https://msdn.microsoft.com/en-us/library/hh279674.aspx) that delete automatically when the program terminates.

Next, you'll see a demonstration of static versus dynamic memory allocation.

<https://stackoverflow.com/questions/162941/why-use-pointers>

**Good insight in this answer, but it does not seem to be 100% accurate (judging by the comments).**

* Why use pointers over normal variables?

Short answer is: Don't. ;-) Pointers are to be used where you can't use anything else. It is either because the lack of appropriate functionality, missing data types or for pure perfomance. More below...

* When and where should I use pointers?

Short answer here is: Where you cannot use anything else. In C you don't have any support for complex datatypes such as a string. There are also no way of passing a variable "by reference" to a function. That's where you have to use pointers. Also you can have them to point at virtually anything, linked lists, members of structs and so on. But let's not go into that here.

How do you use pointers with arrays?

With little effort and much confusion. ;-) If we talk about simple data types such as int and char there is little difference between an array and a pointer. These declarations are very similar (but not the same - e.g., sizeof will return different values):

char\* a = "Hello";

char a[] = "Hello";

You can reach any element in the array like this

printf("Second char is: %c", a[1]);

Index 1 since the array starts with element 0. :-)

Or you could equally do this

printf("Second char is: %c", \*(a+1));

The pointer operator (the \*) is needed since we are telling printf that we want to print a character. Without the \*, the character representation of the memory address itself would be printed. Now we are using the character itself instead. If we had used %s instead of %c, we would have asked printf to print the content of the memory address pointed to by 'a' plus one (in this example above), and we wouldn't have had to put the \* in front:

printf("Second char is: %s", (a+1)); /\* WRONG \*/

But this would not have just printed the second character, but instead all characters in the next memory addresses, until a null character (\0) were found. And this is where things start to get dangerous. What if you accidentally try and print a variable of the type integer instead of a char pointer with the %s formatter?

char\* a = "Hello";

int b = 120;

printf("Second char is: %s", b);

This would print whatever is found on memory address 120 and go on printing until a null character was found. It is wrong and illegal to perform this printf statement, but it would probably work anyway, since a pointer actually is of the type int in many environments. Imagine the problems you might cause if you were to use sprintf() instead and assign this way too long "char array" to another variable, that only got a certain limited space allocated. You would most likely end up writing over something else in the memory and cause your program to crash (if you are lucky).

Oh, and if you don't assign a string value to the char array / pointer when you declare it, you MUST allocate sufficient amount of memory to it before giving it a value. Using malloc, calloc or similar. This since you only declared one element in your array / one single memory address to point at. So here's a few examples:

char\* x;

/\* Allocate 6 bytes of memory for me and point x to the first of them. \*/

x = (char\*) malloc(6);

x[0] = 'H';

x[1] = 'e';

x[2] = 'l';

x[3] = 'l';

x[4] = 'o';

x[5] = '\0';

printf("String \"%s\" at address: %d\n", x, x);

/\* Delete the allocation (reservation) of the memory. \*/

/\* The char pointer x is still pointing to this address in memory though! \*/

free(x);

/\* Same as malloc but here the allocated space is filled with null characters!\*/

x = (char \*) calloc(6, sizeof(x));

x[0] = 'H';

x[1] = 'e';

x[2] = 'l';

x[3] = 'l';

x[4] = 'o';

x[5] = '\0';

printf("String \"%s\" at address: %d\n", x, x);

/\* And delete the allocation again... \*/

free(x);

/\* We can set the size at declaration time as well \*/

char xx[6];

xx[0] = 'H';

xx[1] = 'e';

xx[2] = 'l';

xx[3] = 'l';

xx[4] = 'o';

xx[5] = '\0';

printf("String \"%s\" at address: %d\n", xx, xx);

Do note that you can still use the variable x after you have performed a free() of the allocated memory, but you do not know what is in there. Also do notice that the two printf() might give you different addresses, since there is no guarantee that the second allocation of memory is performed in the same space as the first one.

<https://stackoverflow.com/questions/22146094/why-should-i-use-a-pointer-rather-than-the-object-itself#:~:text=It%20can%20be%20difficult%20to,it%20is%20simpler%20and%20better.&text=C%2B%2B%20gives%20you%20three%20ways,by%20reference%2C%20and%20by%20value.>

Preface

Java is nothing like C++, contrary to hype. The Java hype machine would like you to believe that because Java has C++ like syntax, that the languages are similar. Nothing can be further from the truth. This misinformation is part of the reason why Java programmers go to C++ and use Java-like syntax without understanding the implications of their code.

Onwards we go

But I can't figure out why should we do it this way. I would assume it has to do with efficiency and speed since we get direct access to the memory address. Am I right?

To the contrary, actually. [The heap is much slower](https://stackoverflow.com/questions/2264969/why-memory-allocation-on-heap-is-much-slower-than-on-stack) than the stack, because the stack is very simple compared to the heap. Automatic storage variables (aka stack variables) have their destructors called once they go out of scope.

For example:

{

std::string s;

}

// s is destroyed here

On the other hand, if you use a pointer dynamically allocated, its destructor must be called manually. delete calls this destructor for you.

{

std::string\* s = new std::string;

}

delete s; // destructor called

This has nothing to do with the new syntax prevalent in C# and Java. They are used for completely different purposes.

Benefits of dynamic allocation

**1. You don't have to know the size of the array in advance**

One of the first problems many C++ programmers run into is that when they are accepting arbitrary input from users, you can only allocate a fixed size for a stack variable. You cannot change the size of arrays either. For example:

char buffer[100];

std::cin >> buffer;

// bad input = buffer overflow

Of course, if you used an std::string instead, std::string internally resizes itself so that shouldn't be a problem. But essentially the solution to this problem is dynamic allocation. You can allocate dynamic memory based on the input of the user, for example:

int \* pointer;

std::cout << "How many items do you need?";

std::cin >> n;

pointer = new int[n];

Side note: One mistake many beginners make is the usage of variable length arrays. This is a GNU extension and also one in Clang because they mirror many of GCC's extensions. So the following int arr[n] should not be relied on.

Because the heap is much bigger than the stack, one can arbitrarily allocate/reallocate as much memory as he/she needs, whereas the stack has a limitation.

**2. Arrays are not pointers**

How is this a benefit you ask? The answer will become clear once you understand the confusion/myth behind arrays and pointers. It is commonly assumed that they are the same, but they are not. This myth comes from the fact that pointers can be subscripted just like arrays and because of arrays decay to pointers at the top level in a function declaration. However, once an array decays to a pointer, the pointer loses its sizeof information. So sizeof(pointer) will give the size of the pointer in bytes, which is usually 8 bytes on a 64-bit system.

You cannot assign to arrays, only initialize them. For example:

int arr[5] = {1, 2, 3, 4, 5}; // initialization

int arr[] = {1, 2, 3, 4, 5}; // The standard dictates that the size of the array

// be given by the amount of members in the initializer

arr = { 1, 2, 3, 4, 5 }; // ERROR

On the other hand, you can do whatever you want with pointers. Unfortunately, because the distinction between pointers and arrays are hand-waved in Java and C#, beginners don't understand the difference.

**3. Polymorphism**

Java and C# have facilities that allow you to treat objects as another, for example using the as keyword. So if somebody wanted to treat an Entity object as a Player object, one could do Player player = Entity as Player; This is very useful if you intend to call functions on a homogeneous container that should only apply to a specific type. The functionality can be achieved in a similar fashion below:

std::vector<Base\*> vector;

vector.push\_back(&square);

vector.push\_back(&triangle);

for (auto& e : vector)

{

auto test = dynamic\_cast<Triangle\*>(e); // I only care about triangles

if (!test) // not a triangle

e.GenericFunction();

else

e.TriangleOnlyMagic();

}

So say if only Triangles had a Rotate function, it would be a compiler error if you tried to call it on all objects of the class. Using dynamic\_cast, you can simulate the as keyword. To be clear, if a cast fails, it returns an invalid pointer. So !test is essentially a shorthand for checking if test is NULL or an invalid pointer, which means the cast failed.

Benefits of automatic variables

After seeing all the great things dynamic allocation can do, you're probably wondering why wouldn't anyone NOT use dynamic allocation all the time? I already told you one reason, the heap is slow. And if you don't need all that memory, you shouldn't abuse it. So here are some disadvantages in no particular order:

It is error-prone. Manual memory allocation is dangerous and you are prone to leaks. If you are not proficient at using the debugger or valgrind (a memory leak tool), you may pull your hair out of your head. Luckily RAII idioms and smart pointers alleviate this a bit, but you must be familiar with practices such as The Rule Of Three and The Rule Of Five. It is a lot of information to take in, and beginners who either don't know or don't care will fall into this trap.

It is not necessary. Unlike Java and C# where it is idiomatic to use the new keyword everywhere, in C++, you should only use it if you need to. The common phrase goes, everything looks like a nail if you have a hammer. Whereas beginners who start with C++ are scared of pointers and learn to use stack variables by habit, Java and C# programmers start by using pointers without understanding it! That is literally stepping off on the wrong foot. You must abandon everything you know because the syntax is one thing, learning the language is another.

1. (N)RVO - Aka, (Named) Return Value Optimization

One optimization many compilers make are things called **elision** and **return value optimization**. These things can obviate unnecessary copys which is useful for objects that are very large, such as a vector containing many elements. Normally the common practice is to use pointers to **transfer ownership** rather than copying the large objects to **move** them around. This has lead to the inception of **move semantics** and **smart pointers**.

If you are using pointers, (N)RVO does **NOT** occur. It is more beneficial and less error-prone to take advantage of (N)RVO rather than returning or passing pointers if you are worried about optimization. Error leaks can happen if the caller of a function is responsible for deleteing a dynamically allocated object and such. It can be difficult to track the ownership of an object if pointers are being passed around like a hot potato. Just use stack variables because it is simpler and better.