**C++ Optimization Techniques**

Now comes the practical part of C++ optimization. You are going to learn a handful of code optimization strategies and then apply those strategies to increase the speed of a C++ program.

However, remember that optimizing a program involves other facets besides the programming language itself. Your program's speed will also depend on your hardware, your compiler and what computer algorithms you choose. The more familiar you become with all of these different facets, the more tools you will have for optimization. Here is a brief summary of why each of these aspects is so important.

**Hardware**

Some hardware might have limitations that slow down your code. For instance, when calculating trigonometric functions, a processor might use a slow software approximation. If instead, you could use [**small-angle approximation**](https://en.wikipedia.org/wiki/Small-angle_approximation), you might get your code to run faster.

Embedded hardware might not have much memory or have a 16-bit or 32-bit architecture instead of a 64-bit architecture. Using 64-bit integers on a 16-bit architecture might be possible with your compiler, but it would also probably be inefficient.

**Compilers**

Many compilers will optimize at least parts of your code for you. For instance, it might be more efficient for the CPU to unroll a for loop to avoid checking the conditional statements:

*// for loop*

**for** (**int** i = 0; i < 5; i++) {

std::cout << i << "\n";

}

*// for loop unrolled*

std::cout << 0 << "\n";

std::cout << 1 << "\n";

std::cout << 2 << "\n";

std::cout << 3 << "\n";

std::cout << 4 << "\n";

The unrolled version could run faster because unrolling avoids checking if i < 5 is true. For a more complete list of what your compiler might try to do, read this article [**here**](https://en.wikipedia.org/wiki/Optimizing_compiler).

**Algorithms**

Some algorithms are known to be faster than other algorithms. A common case would be sorting algorithms; [**quicksort**](https://en.wikipedia.org/wiki/Quicksort), for example, is known to be faster than [**bubble sort**](https://en.wikipedia.org/wiki/Bubble_sort).

Here is another important point to keep in mind; C++ libraries are very convenient, but that doesn't mean they use the fastest algorithms especially for your individual case. Being aware of what is happening under the hood provides more opportunities for improving efficiency.

**C++**

And now, let's move on to optimizing C++. You are going to learn a handful of techniques and practice implementing them. Then at the end of the lesson, you'll have the opportunity to optimize a C++ histogram filter.

Here is a preview of the type of things you will be learning: Did you know that every time you call a function, C++ copies the input variables into memory? Take this example:

**#include <iostream>**

**int** **addition**(**int** a, **int** b);

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

**int** x, y;

x = 5;

y = 7;

std::cout << addition(x, y) << "\n";

}

**int** **addition**(**int** a, **int** b) {

**return** a + b;

}

C++ puts the x variable into memory and the y variable into memory as expected.

When you call the addition function, C++ actually then puts the a variable and the b variable into memory as well; essentially C++ is copying x and y into memory twice even though the x and y values could have been used directly.

For a 32-bit integer, this might not be an issue; however, once you start working with larger variables such as 2-D vectors, the extra read and writes can slow your programs down.

In this lesson, you'll learn how to speed up your code in situations like these.

## Software Development

Writing functioning code is arguably your number one goal as a software developer. Depending on your application, code efficiency might be very important as well.

You could break down the code development process into the following steps:

* code design
* implementing the design
* testing for bugs and fixing the bugs
* optimization

For a more detailed explanation of each of these steps (design, implementation, testing), see this [**link**](https://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Optimization_life_cycle).

As mentioned previously, there are many facets to code optimization related to hardware, compilers, algorithms and the C++ language itself. In this lesson, you'll focus on the C++ language. What you've just learned about the CPU and RAM will provide the context for why your code is able to run faster.

You will have the opportunity to learn and implement a few techniques that make C++ run even faster. Each exercise presents a "slow" version of the code and a technique for making the code faster. You will then implement the technique to see how much faster the code runs. While these techniques do not encompass all of the ways to optimize C++ code, you'll gain an understanding of how memory, the CPU and coding choices affect how fast your code runs.

At the end of the lesson, you are going to receive a set of files for a functioning C++ histogram filter. Your job will be to make the histogram filter code run faster using the techniques you've learned in the lesson.

## How to Optimize: Testing versus Instinct

To optimize your code, rely on testing and verification rather than instinct!

Test your code to find areas that are inefficient in terms of time, memory or power use. Then, verify that any changes you have made really do make the code more efficient. If you test first, you might even find that your code is already efficient enough for your particular application.

If you were optimizing a large amount of code, you would want to use something called a profiler. A profiler is a piece of software that measures how long parts of your code are taking to execute or how many resources the code uses. The profiler helps you find congestion points so that you can optimize the least efficient parts of the code first.

Both Visual Studio and Xcode come with profilers, which you can read about at these links:

* [**Profiling in Visual Studio**](https://docs.microsoft.com/en-us/visualstudio/profiling/beginners-guide-to-performance-profiling)
* [**Profiling in Xcode**](https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/index.html)

### Testing your Code with Standard Clock

In the following exercises, we've set up some simple profiling code for you; you'll time how long it takes to run a function using the C++ standard clock.

You will optimize by:

* seeing how long it takes to run a function
* change some aspect of the code
* run the code again to see if the code runs faster

The profiling code has already been set up for you, and it looks like this:

**#include <ctime>**

std::**clock\_t** start;

**double** duration;

start = std::clock();

function\_name(var1, var2);

duration = ( std::clock() - start ) / (**double**) CLOCKS\_PER\_SEC;

std::cout << "duration milliseconds initialize beliefs " << 1000 \* duration << '\n';

The code stores the current standard clock time, then runs a function, and then calculates the elapsed time.

The standard library clock measures time in terms of **clock ticks**. Clock ticks are like a counter variable that goes up at a constant rate of time. But the length of time varies from system to system; therefore, dividing by the constant CLOCKS\_PER\_SEC gives the results in terms of seconds. The CLOCKS\_PER\_SEC variable is defined in the ctime library.

# Overview of Techniques in this Lesson

Here is an overview of the techniques you will be learning and using to optimize C++ code. Refer back to this page as you work through the exercises and on the final project. Each of these techniques will be expanded further throughout the lesson.

If you are wondering why we chose these techniques, it's because you can use all of them to optimize Andy's histogram filter code. This is not a comprehensive list of optimization techniques by any means; rather, practicing these techniques will get you thinking about your code from a new perspective. While your focus up until now has been on code implementation, now you will appreciate the consequences of your coding decisions.

## Remove Dead Code

When you write your code, you might end up with a chunk of code that is no longer used but still part of the program. Watch out for cases like these: remember that every line of code involves some work by the CPU and oftentimes a read or write to memory. Any unnecessary code could slow things down.

## Avoid Extra if Statements

Code with multiple branching if statements is oftentimes inefficient:

**int** x = 5;

**if** (x >= 5) {x = x + 1;}

**if** (x < 5) {x = x - 1;}

You could transform the above code into if else statements, which avoids the CPU checking whether x is less than 5.

## Avoid Nested for Loops

A section of code like this:

**for** (**int** i = 0; i < 5; i++) {

**for** (**int** j = 0; j < 4; j++) {

matrix[i][j];

}

}

should look somewhat familiar. This is how you have been iterating over the values in a matrix. Sometimes you have no choice but to use a nested for loop; however, depending on the application, you might not need a nested for loop if you think about the problem in a different way. And instead of needing twenty iterations like in the above code, you might only need 9 or fewer iterations.

## Avoid Creating Extra Variables

You'll see that Andy sometimes creates extra variables that aren't needed. For example,

**float** x = 2;

**float** y = 7;

**float** z = 4;

**float** volume = x \* y \* z;

**float** volume\_reciprocal = 1 / volume;

The volume variable isn't really necessary. You could calculate the reciprocal directly:

volume\_reciprocal = 1 / (x \* y \* z);

You might not get much of a performance boost; float variables are relatively efficient in C++. But imagine creating a new variable to hold a larger variable like a 2D vector. All of the extra memory writes will definitely slow things down.

## Reserve Space in Memory for Vectors

You've been using 2D vectors to represent matrices. C++ vectors have a big benefit in terms of convenience; you can add new elements to the vector as needed. Arrays, on the other hand, have a fixed length that cannot be changed after declaring them.

But this flexibility comes at a cost; C++ vectors are very inefficient in terms of execution time. In fact, programs that require speedy execution would generally not use C++ vectors and especially not big for loops to iterate through vectors; instead, programs are written to take advantage of [parallel processing](https://en.wikipedia.org/wiki/Parallel_computing) on either the CPU or a GPU (graphics processing unit).

When you declare and define a vector, the compiler reserves space in memory plus some extra bytes in case the vector expands. Once the vector's length expands past the reserved memory, the entire vector will get copied over to a different place in RAM with enough available space.

That is very inefficient! In Andy's histogram filter code, you already know how large your vectors need to be because the robot world has a fixed number of grid spaces. If you reserve space for the vector, then you can avoid all of the extra memory reallocations as you expand the vector's length. The syntax is as simple as coding:

std::vector<**int**> foo;

foo.reserve(15);

Now the foo vector is guaranteed to have enough space for holding fifteen integers.

## Passing Variables by Reference

Whenever you call a function, C++ copies any input variables into memory even if those variables are already in memory. For fundamental data types like int, char, or float, this might not be a problem.

But with variables that take up a more significant amount of space, such as vectors, the extra copying can slow down your programs.

You'll learn how to pass variables by reference instead of by value. Passing by reference tells your function to use the variable directly in memory rather than copying the entire variable to memory over again.

## Using Static

In Andy's code, you'll see that he calculates certain vectors and values inside a function, but these values are always the same every time the function is called.

Instead of calculating the variables over and over again, you can declare these variables as static. When the function is called the first time, C++ stores the values in memory and re-uses the values every time the function is called.

## Next Part of the Lecture

The next part of the lessons has a series of exercises that will get you ready for the project. In each exercise, you will receive code that has one of the issues discussed on this page. And then you will change the code to get the program to run faster.

You'll see that every time you run the code, the timer gives slightly different results; when you run your C++ program, the CPU might be carrying out other tasks that could affect timing. Be sure to execute your code multiple times to convince yourself that your code is (or isn't) running faster.

# What is Dead Code?

Dead code is extra code in your program that no longer serves a purpose. Maybe while implementing a solution, you started down a path that did not end up working out. Or you put in extra code for debugging purposes, but this code is not actually part of the solution. The dead code just sits in your program without affecting the implementation.

Dead code can slow down a program; the code will still be executed on the CPU and might include reads and writes to memory although the code is never used.

Two other issues related to dead code, and sometimes also considered to be dead code, are **redundant code** and **unreachable code**.

### Redundant Code

Redundant code is just like it sounds; this is code that gets repeated multiple times although isn't necessary. Here's a quick example:

*// example of redundant code*

**int** x = 6;

**if** (x > 5) {

**return** true;

}

**else** {

**return** false;

}

Essentially, saying x > 5 and then returning True is redundant. You could eliminate the if statement by just saying

**return** x > 5;

Here's another example

**if** (x < 5) {

x = x + 1;

}

**else** **if** (x >= 5 && x < 10) {

x = x + 2;

}

There is no need to check if x >= 5; the first if statement has essentially already proven whether x is greater than or less than five.

Finding redundant code, however, might not be so easy. It could be an extra variable declaration or maybe some extra logical statements like in the above example.

## Unreachable Code

Unreachable code never gets executed. Unreachable code might not have too much of an effect on code speed, but the code still takes up space in memory and can make memory management less efficient.

A simple example would be the following:

**unsigned** **int** x;

...

**if** (x >= 0) {

do\_something ..

}

**else** {

do\_something ..

}

An unsigned integer is always zero or positive, so the else statement will never execute. The code inside the else statement is unreachable.

When you get to Andy's code at the end of the lesson, just keep an eye out for dead code or redundant code. You might find a couple of spots.

In the next part of the lesson, you will see a program that adds two matrices together. You'll execute the code to see how fast it runs. But the program contains dead code. So you will remove the dead code to see if the program runs faster.

# If Statements

Many compilers will try to optimize if statements for you. So writing something like this:

**int** x = 7;

**if** (x > 0) {

**return** y;

**if** (x <= 0) {

**return** z;

}

might become

**int** x = 7;

**if** (x > 0) {

**return** y;

**else** {

**return** z;

}

The compiler won't literally rewrite your C++ code into optimized C++ code; the compiler does the optimization when outputting assembly language or machine code. But in general, you'll want to avoid too many if branches because each branch's logical expression requires time on the CPU.

Thinking about how if statements are executed, it's also more efficient to put the most common cases on a higher branch.

Here is a simple example:

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

**if** (i > 0 && i < 5) {

cout << "low \n";

}

**else** **if** (i >= 990) {

cout << "high \n";

}

**else** {

cout << "normal \n";

}

}

Most of the time, the above code will print out the word "normal". So the code needs to go through all of the if and else branches most of the time as the CPU compares **i** in each branch.

It would be more efficient to put the "normal" case at the top of the branches instead of at the bottom:

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

**if** (i >= 5 && i < 990) {

cout << "normal \n";

}

**else** **if** (i >= 990) {

cout << "high \n";

}

**else** {

cout << "low \n";

}

}

Now, most of the time, the CPU can skip the else if and else branches.

# If Statements on the CPU

There's one other aspect of if statements that you don't have much control over when using a high level language like C++. The CPU also tries to optimize if statement calculations by running simultaneous calculations.

When running a calculation, the CPU can look ahead and start working on another calculation in parallel. In terms of if statements, the CPU will try to predict which branch will be taken next and starts running the calculations inside the predicted branch. When it's time to evaluate the logical expression, the CPU might realize that it made a bad prediction. If the prediction is wrong, the predicted calculation stops and the CPU starts running the correct calculation.

So be aware that you might not get too much of a time boost when rearranging or eliminating if statements. Both the compiler and the CPU are already trying to optimize these operations for you.

# Exercise

Move on to the next section to try out an if statement optimization exercise.

# For Loops

There is nothing wrong with using nested for loops (ie for loops inside of for loops). Sometimes you need them when working with C++ vectors.

However, don't use them if you don't need them! There are a few places in Andy's code where he has used nested for loops that were not needed.

If you are iterating through or initializing an m by n matrix, you might be tempted to always use nested for loops like this:

**for** (**unsigned** **int** i = 0; i < matrix.size(); i++) {

**for** (**unsigned** **int** j = 0; j < matrix[0].size(); j++) {

**do** something...

}

}

Iterating through the entire matrix involves m times n operations. However, depending on what you are trying to do, you might be able to get away with doing something like this:

**for** (**unsigned** **int** i = 0; i < matrix.size(); i++) {

**do** something

}

**for** (**unsigned** **int** j = 0; j < matrix[0].size(); j++) {

**do** something

}

This only requires m + n operations instead of m \* n operations. Remember that fewer instructions for the CPU will get your code to execute faster!

To see a specific example, go to the for loops exercise in the next part of the lesson.

# Exercise

In the next section, you'll find an exercise for optimizing a for loop. The code initializes a 2D vector using a for loop inside another for loop, which uses m\*n operations. Think about how you could initialize the same 2D vector using only m + n operations (hint: remember that every position in the 2D vector has the same value).

# Intermediate Variables

In Andy's code, you'll notice that he sometimes uses intermediate variables. Intermediate variables could be considered redundant code.

For example:

**float** x = 5.8;

**float** y = 7.1;

**float** area = x \* y;

**float** reciprocal\_area = 1/(area);

If you only needed to calculate the reciprocal, you could have written:

**float** x = 5.8;

**float** y = 7.1;

**float** reciprocal\_area = 1/(x\*y);

Fundamental variable types like float, int, and char are relatively efficient. So you probably won't notice much of a different when running these two versions of the code. In fact, your compiler might end up eliminating any inefficiencies between the first version and the second version.

So why discuss intermediate variables?

# Intermediate Matrix Variables

You'll find a spot in Andy's code where he actually uses a 2D vector as an intermediate variable. It's not as obvious as the above example. But if you find it, you can definitely make the code run faster by eliminating the extra variable.

As mentioned in the beginning of the lesson, vectors are convenient but not particularly efficient; the compiler allocates a certain amount of memory for a new vector and adds a few more bytes as a buffer. The buffer can hold extra elements that you might push to the back of the vector.

But when the vector increases beyond its allocated size, the whole entire vector gets copied to another part of RAM. That is super inefficient!

So if you already have a variable vector that you can update directly, avoid making a copy of the vector!

# Exercise

In the next exercise, you'll get to see just how inefficient it is to create new vectors. The code iterates through a 2-dimensional vector dividing each element by a constant. But instead of iterating the vector directly, the code creates a new, but unnecessary 2-D vector.

**Vector Memory Storage**

As previously mentioned, vectors are not the most efficient variable type in C++. One reason is because you do not need to specify a vector's length when declaring a vector variable. So the compiler will not know ahead of time how much memory to allocate. And once the vector's length goes beyond the initial allocated memory, the entire vector gets copied to a part of RAM with more space.

A vector is more efficient if you specify the vector's length before pushing values. You can do this with the reserve() method, which will guarantee that the vector can hold the number of elements reserved.

**Example**

Here is an example of how to use the reserve() method.

**#include <iostream>**

**#include <vector>**

**using** **namespace** std;

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

vector<**int**> myvector;

**int** vector\_size = 50;

myvector.reserve(vector\_size);

**for** (**int** i = 0; i < vector\_size; i++) {

myvector.push\_back(i);

}

**return** 0;

}

In C++, there are multiple ways to initialize a 2D vector. When optimizing your programs, you'll need to test out different ways to initialize to see which works best in your specific program.

The fastest way could end up depending on the vector length and vector types.

**Exercise**

In this next exercise, you're given three different functions in three separate files:

* reserved.cpp
* unreserved.cpp
* initializer.cpp

Your task will be to change the code in reserved.cpp so that the 2D vector variable and the 1D vector variable reserve their sizes before using the push\_back method. You'll see if that change helps to make the code run faster.

The unreserved() function will initialize a 2D vector using a for loop. The initializer() function uses a standard library method for initializing a vector. The code in reserved.cpp is exactly the same as the code in unreserved.cpp. Your job will be to changed reserved.cpp to use the reserve() method.

# References

Did you know that every time you call a function, those input variables get copied into memory? You can prove this to yourself with the following code:

**#include <iostream>**

**#include <vector>**

**using** **namespace** std;

*// function declaration that prints out a matrix's address in memory*

**void** **matrix\_address**(vector< vector<**int**> > my\_matrix);

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

*// initialize a matrix*

vector< vector<**int**> > matrix(5, vector<**int**>(6,2));

*// print out the matrix address*

cout << "original variable address: " << &matrix << "\n";

*// run a function that prints out a matrix address*

matrix\_address(matrix);

**return** 0;

}

*// function to print out a matrix address*

**void** **matrix\_address**(vector< vector<**int**> > my\_matrix) {

cout << "function variable address: " << &my\_matrix << "\n";

}

When you run this code, you'll get output that looks something like this:

**original variable address: 0x7fff5fbff650**

**function variable address: 0x7fff5fbff608**

So what is this code doing? The code initializes a 5x6 2D vector in a variable called matrix. Then the code prints out the address in memory where the 2D vector starts.

Next, the code calls a function that prints out the address in memory of the function's input variable. Notice that the addresses are not the same even though the two variables hold the same value.

This is because C++ is copying the 2D vector into memory again when you call the matrix\_address() function.

# Ampersand Symbol

In the above code, you might have noticed the ampersand symbol: &.

This symbol gives you the address of a variable rather than the value of a variable. Do you remember in the previous lesson when you briefly learned about pointers? The ampersand is an easy way to access a variable's address without the danger of you mistakenly messing something up in memory.

And you can use the ampersand to help speed up your code!

# How the Ampersand Can Speed Up Code

C++ has a few fundamental data types like int, char, and float that are relatively fast to work with. So the coding strategy you're about to learn might not make much of a different with fundamental data types; however, with variables that take up a lot of memory such as arrays or vectors, the ampersand can be quite useful.

What if the matrix\_address function shown above had been defined like this?

**void** **matrix\_address**(vector< vector<**int**> >&my\_matrix);

The ampersand (&) tells the compiler to pass the input variable by reference. That means inside your function, you'll be working with the original variable instead of a copy. In the case of a 2D vector, you've just saved yourself a lot of reads and writes.

And, depending on your application, you might be able to modify the input vector directly instead of creating a new vector inside your function. For example, if you were going to code scalar multiplication on a vector, and you didn't need to keep the original vector, you could modify the original vector directly. You'll see what this means in the exercise.

# Exercise

In this exercise, you're going to go back to the matrix addition function presented earlier in the lesson. You'll rewrite a version of the code passing matrices by reference. Let's see if this gets the code to run faster!

# The Static Keyword

Sometimes you'll need to use the same variable over and over again in your functions. If you think back on Andy's Python histogram filter code, the blur() function was an example.

The blur function contains a handful of blurring factors that are always the same every time the function gets called; however, in Andy's C++ code, which you'll be working with soon, he recalculates these blurring factors every time the code gets run.

What if you could declare and define these variables once no matter how many times your function gets called? You'd be able to eliminate some reads and writes to memory. This is the perfect use case for the C++ static keyword.

# Example

When you declare and define a variable inside a C++ function, the value gets allocated to memory.

For example,

some\_function() {

**int** x = 5;

}

allocates space in memory for the variable x and then assigns the value five. Then, when the function finishes, the CPU will remove the x variable from RAM. That means every time you run the function, the CPU will allocate and deallocate memory for the x variable.

If, on the other hand, your code uses the static keyword, the x variable gets allocated to memory the first time the function runs. And the x variable just stays allocated in memory for the duration of the entire program. You've just saved yourself some reads and writes to RAM:

some\_function() {

**static** **int** x = 5;

}

Notice that you need to declare and define the variable simultaneously. You cannot define a variable with the static keyword without giving the variable a value.

# Global Variables versus Static Variables

Static variables are actually placed in the same area of RAM as global variables. The difference is that global variables are declared outside of functions and are available anywhere in your program to any file or function. On the other hand, static variables remain in scope. So in the above example, some\_function() is the only place that can access the x variable.

# Exercise

In this exercise, you'll see how much faster code can run when using the static keyword. You'll also have an opportunity to eliminate some for loops!

Think back on how the blur() function worked in the Python histogram filter lesson. You had a 2D grid representing the robot's world. Each grid space had a probability representing the likelihood that the robot was on that square. After the robot used its sensors and updated its beliefs, a blur function smoothed out the probability values.

In this exercise, you'll be working with a part of Andy's blur function: more specifically the part that calculates the blur window. You'll eliminate some for loops and also use the static keyword to get the code to run faster.