C++ InfoPack

|  |  |  |  |
| --- | --- | --- | --- |
| Kemal Daysal |  | 24.09.2024 |  |

Table of Contents

[Key Tips 5](#_Toc180184326)

[Declaring Variables 5](#_Toc180184327)

[Initial Variable Values 5](#_Toc180184328)

[New and Useful Ways of Declaring Variables after C++11 5](#_Toc180184329)

[List Initialization and it’s Benefits 5](#_Toc180184330)

[Defining Same Variable Both Locally and Globally 5](#_Toc180184331)

[Data Types 5](#_Toc180184332)

[Character Types based on sizes 5](#_Toc180184333)

[Integer Types 5](#_Toc180184334)

[Float Types 5](#_Toc180184335)

[Bool 5](#_Toc180184336)

[Sizeof Operators and Climits & Cfloat Libraries 5](#_Toc180184337)

[Constants and No-More Usage of Defines 5](#_Toc180184338)

[Arrays and Vectors 5](#_Toc180184339)

[Arrays 5](#_Toc180184340)

[Tips and Useful Information 5](#_Toc180184341)

[Multidimensional Arrays 5](#_Toc180184342)

[Coding Info 5](#_Toc180184343)

[Initial Values of New Declared Arrays 5](#_Toc180184344)

[Multidimensional Arrays 5](#_Toc180184345)

[Vectors 5](#_Toc180184346)

[Coding Info 5](#_Toc180184347)

[Declaring Vectors 5](#_Toc180184348)

[Initializing Values to Declared Vectors 5](#_Toc180184349)

[Accessing Vector Elements 5](#_Toc180184350)

[Changing the Contents of Vector Elements 5](#_Toc180184351)

[When do they change size dynamically (Pushback Method)? 5](#_Toc180184352)

[What happens if you are out of bounds? 5](#_Toc180184353)

[Removing Selected Element from Vector (Erase Function) 5](#_Toc180184354)

[Multidimensional Vectors 5](#_Toc180184355)

[Why sizeof operator doesn’t work in vectors and what to use instead? (.size) 5](#_Toc180184356)

[Assignment Operator 5](#_Toc180184357)

[Arithmetic Operations 5](#_Toc180184358)

[Modulo/Remainder (%) Operator 5](#_Toc180184359)

[Dividing two integers to get a float result 5](#_Toc180184360)

[Processing Priority (PEMDAS) Rule in Math 5](#_Toc180184361)

[Increment – Decrement Operators 5](#_Toc180184362)

[Pre-Increment 5](#_Toc180184363)

[Post-Increment 5](#_Toc180184364)

[Mixed Type Expressions and Conversions 5](#_Toc180184365)

[Mixed Type Expressions 5](#_Toc180184366)

[Explicit Type Casting – static\_cast<type> 5](#_Toc180184367)

[Testing for Equality 5](#_Toc180184368)

[Comparing two doubles too close to each other, then fails 5](#_Toc180184369)

[Comparing one integer to one double (mixed-mode comparison) 5](#_Toc180184370)

[Relational Operators 5](#_Toc180184371)

[Three-way Comparison Operator (<=>) 5](#_Toc180184372)

[Logical Operators 5](#_Toc180184373)

[Precedence (Öncelik) 5](#_Toc180184374)

[Short-Circuit Evaluation 5](#_Toc180184375)

[Compound Assignment 5](#_Toc180184376)

[Operator Precedence 5](#_Toc180184377)

[If-Else Statements 5](#_Toc180184378)

[Semi-Colon after if statement 5](#_Toc180184379)

[Switch-Case Statements 5](#_Toc180184380)

[Main Overview 5](#_Toc180184381)

[Performance comparison with if statements 5](#_Toc180184382)

[Enumeration Input with Switch Case 5](#_Toc180184383)

[Benefits 5](#_Toc180184384)

[One Statement for Multiple Cases 5](#_Toc180184385)

[If-Else Statement inside Switch-Case 5](#_Toc180184386)

[Conditional Operator 5](#_Toc180184387)

[Looping 5](#_Toc180184388)

[For Loop 5](#_Toc180184389)

[Basic Tips 5](#_Toc180184390)

[Multiple criterias and multiple operations in a single for loop 5](#_Toc180184391)

[Using doubles in loop control 5](#_Toc180184392)

[Endless for loop 5](#_Toc180184393)

[Range based for loop 5](#_Toc180184394)

[“Auto” keyword 5](#_Toc180184395)

[Providing collection right in the loop 5](#_Toc180184396)

[Using it on a string 5](#_Toc180184397)

[Do While 5](#_Toc180184398)

[Continue and Break Statements 5](#_Toc180184399)

[Continue 5](#_Toc180184400)

[Break 5](#_Toc180184401)

[Examples 5](#_Toc180184402)

[Collection checker with exit and skip conditions 5](#_Toc180184403)

[Infinite Loops 5](#_Toc180184404)

[Overview 5](#_Toc180184405)

[Examples 5](#_Toc180184406)

[Infinite for loop 5](#_Toc180184407)

[Infinite while loop 5](#_Toc180184408)

[Infinite do-while loop 5](#_Toc180184409)

[Quit Event Checker 5](#_Toc180184410)

[Nested Loops 5](#_Toc180184411)

[Examples 5](#_Toc180184412)

[Multiplication table from 1 to 10 (10x10) 5](#_Toc180184413)

[Filling and Displaying 2D Arrays 5](#_Toc180184414)

[Nested Loops with Vectors 5](#_Toc180184415)

[Creating a dynamic vector and displaying it’s elements as a histogram 5](#_Toc180184416)

[Characters and Strings 5](#_Toc180184417)

[Character Functions 5](#_Toc180184418)

[Functions – Character Tests 5](#_Toc180184419)

[Functions – Character Conversions 5](#_Toc180184420)

[C-style Strings 5](#_Toc180184421)

[Declaring and Changing Strings 5](#_Toc180184422)

[Functions that work with c-style strings 5](#_Toc180184423)

[Cstdlib Library 5](#_Toc180184424)

[Examples and Common Mistakes 5](#_Toc180184425)

[Proper way of assigning a value containing a whitspace to a string 5](#_Toc180184426)

[C++ Strings 5](#_Toc180184427)

[Declaring and initializing 5](#_Toc180184428)

[Assignment 5](#_Toc180184429)

[Concatenation 5](#_Toc180184430)

[Accessing characters with [] and at() method 5](#_Toc180184431)

[Comparing 5](#_Toc180184432)

[Substrings – substr() 5](#_Toc180184433)

[Searching – find() 5](#_Toc180184434)

[Removing characters – erase() and clear() 5](#_Toc180184435)

[Length Method 5](#_Toc180184436)

[Compound Concatenation Assignment 5](#_Toc180184437)

[Input >> and getline() 5](#_Toc180184438)

[Insertion Operator - .insert 5](#_Toc180184439)

[Swap Operator - .swap 5](#_Toc180184440)

[Functions 5](#_Toc180184441)

[Using Functions of Some Popular Libraries 5](#_Toc180184442)

[Cmath Library 5](#_Toc180184443)

[Random Number Generation with <cstdlib> 5](#_Toc180184444)

[Declaring Functions & Function Prototypes 5](#_Toc180184445)

[Function Parameters and Outputs 5](#_Toc180184446)

[Pass by Value 5](#_Toc180184447)

[Formal Parameters 5](#_Toc180184448)

[Actual Parameters (Arguments) 5](#_Toc180184449)

[Function Return Statement 5](#_Toc180184450)

[Default Argument Values 5](#_Toc180184451)

[Downside of Default Argument Values 5](#_Toc180184452)

[Overloading Functions 5](#_Toc180184453)

[Restriction of Function Overloading 5](#_Toc180184454)

[Output Error 5](#_Toc180184455)

[Overloading with Default Arguments 5](#_Toc180184456)

[Passing Arrays to Functions 5](#_Toc180184457)

[A Potential Issue (Changing Original Array) 5](#_Toc180184458)

[const parameters 5](#_Toc180184459)

[Passing Vectors into Functions 5](#_Toc180184460)

[Pass by Reference 5](#_Toc180184461)

[Advantages and Use-Cases 5](#_Toc180184462)

[More Examples 5](#_Toc180184463)

[Potential Issues and Undesired Cases 5](#_Toc180184464)

[Scope Rules 5](#_Toc180184465)

[Static Local Variables 5](#_Toc180184466)

[Global Scope and Global Identifiers, Global Variables 5](#_Toc180184467)

[Examples 5](#_Toc180184468)

[Creating and Reviewing Different Scopes 5](#_Toc180184469)

[Global Variable Basics Example and Changing It 5](#_Toc180184470)

[Static Local Variable Basics Example 5](#_Toc180184471)

[How do Function Calls work in background? 5](#_Toc180184472)

[What happens when main function calls another function? 5](#_Toc180184473)

[Inline Functions 5](#_Toc180184474)

[Recursive Functions 5](#_Toc180184475)

# Key Tips

## Declaring Variables

### Initial Variable Values

If you don’t give a initial value,

* While local variables can take any random value,
* Global variables are automatically initialized to 0, if not initialized by user.

### New and Useful Ways of Declaring Variables after C++11

#### List Initialization and it’s Benefits

Instead of using

* int num = 0;
* int num {0};
  + is a new and modern way to declare a parameter, which is list initialization.
* List initialization prevents data size overflows in when compiling and gives a compiler error, preventing the execution while the C-type one doesn’t give an error, just giving a warning and then generating an unexpected negative value when the program is executed.
* Warning: It doesn’t give overflow error when fitting two variables’ sum, multliplication etc. to a inappropriate size. It’ll just give a warning and let the user to execute the program and then it could display an unexpected negative value as the result. Be aware of the warnings.

A number with red and blue text

Description automatically generated

### Defining Same Variable Both Locally and Globally

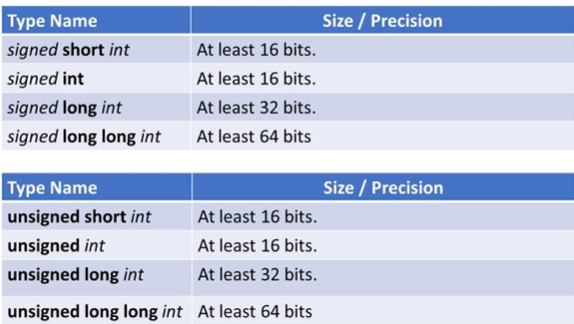
* When a variable is both declared globally and locally, then if this variable needs to be used inside of a function, the compiler decides to first look at the closer declaration to it. If the variable is defined in the function, compiler ignores the global-defined variable and then uses the local-defined one.
* Avoid using global variables as much as possible and define them locally if possible.

## Data Types

### Character Types based on sizes

* char: exactly one byte, at least 8 bits. It can represent 256 characters, which is enough for representing every letter of many spoken languages like Latin alphabet.
* However, many spoken languages have thousands of characters, that can be as large as necessary to represent these characters. In this case, we can use below:
* char16\_t : At least 16 bits
* char32\_t : At least 32 bits
* wchar\_t : Can represent the largest available character set

### Integer Types



* Instead of using these above, isn’t using something like uint16\_t or int32\_t is more clear, easy to remember and understandable?

A screenshot of a computer code

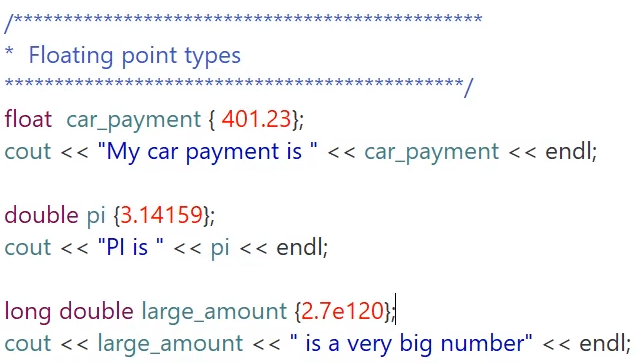
Description automatically generated

* After C++14, we can declare long numbers with single quote marks to help the programmer to read the code easily. C++14 removes them when compiling and displays/uses the exact number without quotes. It’s just there for helping the programmers.

### Float Types

A table with numbers and a few words

Description automatically generated with medium confidence



### Bool

* Bool is usually 8 bits

## Sizeof Operators and Climits & Cfloat Libraries

* At first, don’t forget that sizes depend on the machine and 32-64 bit systems.
  + Even the sizes differ when running a 32-bit compiler on a 64-bit OS compared to 64-bit compiler on a 64-bit OS.
* Climits provides information about integral types
* Cfloat provides information about floating point types
* These libraries also provide a bunch of handy defined constants that we can use to determine the precision of our primitive types. For example:
  + INT\_MAX tells us the maximum value we can storoe in an int on this specific machine.
  + INT\_MIN
  + LONG\_MIN
  + LONG\_MAX
  + FLT\_MIN
  + FLT\_MAX etc.
* After including <climits> or/and <cfloat> depending on the need, general usage is like this:
  + sizeof(char)
  + sizeof(long double)
  + sizeof(my\_age)
  + cout << “Minimum value of a char: ” << CHAR\_MIN << endl; (outputs -128 because the char is 1 byte big (2^8 = 256 bits, -128-0-128)
  + cout << “Maximum value of a long long: ” << LLONG\_MAX << endl;

## Constants and No-More Usage of Defines

* Don’t use “defined” constants, defined by preprocessor directive in Modern C++. They were commonly used in legacy C++,
* #define means wherever it sees the word “pi”, replaces it with 3.1415926 (like a macro).
  + Think of this as a blind find-replace, as you might do in a word processor without the need of knowing C++. The preprocessor will gladly substitute one value for another value.
    - But since the preprocessor doesn’t know C++, it can’t type check it. And it could lead to difficult to find errors.
    - Therefore, please don’t use defined constants in modern C++ code.
* Don’t use like below anymore:
  + #define pi 3.1415926
* Instead, use this:
  + const pi {3.1415926};

## Arrays and Vectors

* Multiple dimensional arrays can be very useful for modeling grids, boardas and much more complex data types.
* Vectors are better than arrays in so many ways.
  + Their advantages will be discussed.
* In Modern C++, we rarely use raw arrays. Instead we use vectors.
  + But we should learn arrays since there’s so much legacy C++ code with arrays.
  + And conceptually, arrays are easier to understand than vectors.

### Arrays

#### Tips and Useful Information

* Arrays are fixed in size. Once you create an array of a specific size, the size can’t change.
* Also, arrays will not grow or shrink on their own while the program is running
* So if you create an array that can hold 100 “same data type” variables, then you need space more than 100, you’d have to go back to source code, change the max size and recompile.
* Arrays are stored contiguously (bitişik) in memory.
  + So if you tell the compiler that you need an array of 100 integers, the compiler will generate code that allocates exactly 100 integers that are contiguous in memory.
  + That means the memory will be allocated as one chunk.
* Keep in mind that C++ arrays are not bounds checked.
  + There’s no check to make sure that when you access an individual array element that it’s between initial element (0) and last element (size-1).
  + It’s the programmer’s responsibility to be sure that you don’t go over bounds.
    - If you go over bounds, the program may have strange behaviour and could even crash.
* Always initialize the arrays, just like initializing the variables.
  + Otherwise, they’ll contain unknown values.
* Arrays are extremely efficient in C++ since they are bare bones raw arrays.
  + While C++ vectors are super powerful, dynamic and safer to use than arrays.
* In C++, the name of the array represents the address in memory of the first element in that array. That is the element with index 0.
  + When we access an array element using the subscript operator and provide an index In the square brackets, the compiler knows that you want the element that is at that offset from the beginning of the array.
  + So all the compiler has to do is a pretty simple calculation. For example, if you want the element at index 2, all the compiler has to do is go 2 integers (or 2 doubles, depending on the data type) away from the beginning of the array. The compiler uses sizeof operator to determine where te element will be.
  + This is one reason why raw arrays in C++ are so efficient.
  + But remember, there’s no bounds checking so be careful. If you have an array of 5 integers and you ask the compiler for the integer at the 10th index, it’ll gladly do the calculation and give you whatever happens to be at that location, which can be dangerous.

##### Multidimensional Arrays

* You can use as many dimensions as they’re supported by your system.
  + While there’s no real limit to the number of dimensions you can have for arrays, be aware that some compilers place limits on this but the limits are quite high and you’re unlikely to reach them.

#### Coding Info

##### Initial Values of New Declared Arrays

* When declaring arrays with initial values, we can declare them like this with list initialization:
  + int test\_scores [5] {100, 95, 99, 87, 88};
* To init some known elements to known values then remaining ones to 0
  + int high\_score\_per\_level [10] {3,5};
* To init all values to 0 automatically and faster without a loop.
  + double hi\_temperatures [365] {0};
* To automatically determine the size based on the initialized elements of the array
  + int auto\_sized\_array [] {1,2,3,4,5};

##### Multidimensional Arrays

* The syntax for declaring multidimensional arrays just involves adding another set of square brackets with the new dimension inside them.
* int movie\_rating [3][4] {}; // initialize 3 (rows) x 4 (columns) array filled with 0s
* int movie\_rating\_2 [3][4] // initialize 3x4 array, filled with these values
* {
* {1,2,3,4},
* {5,6,7,8},
* {9,10,11,12}
* };
* For 3 dimensional array
* int movie\_rating\_3d [3][4][2] {};// 3 dimensional array

A screenshot of a computer program

Description automatically generated

### Vectors

* To understand the concept with an example, suppose we want to store test scores for students in a class but registration hasn’t started yet so we don’t know how many students we’ll have in class.
* Also students drop and add as the semester progresses.
* How can we model this information using an array as we’ve seen to this point?
  + Remember, C++ arrays are fixed in size. So we need to specify a size when we declare them. So our options are:
    - Create a static array and pick a size that we’re not likely to exceed, which must be large enough.
      * Many times this option will be okay but we’ll probably make it too big and waste space and memory.
      * For example, we may have 30 students but we allocated 50 students in the array.
      * Not only have we wasted storage space but now we need to keep track of how many students are actually in the array.
      * And of course sooner or later, we’ll get that one student that exceeds the size of the array.
    - We can also create a dynamic array such as a vector to solve some of the problems above.
* A C++ vector is part of the C++ Standard Template Library.
  + STL library contains powerful containers, algorithms, functions and iterators.
  + This means that we have available to us pre-written, pre-tested, easy to use components that we can use to help us solve problems.
    - We’ll go over the main components of the STL toward the end of this course.
* We talked about C++ being a OOP language. When we create a C++ vector, we are creating a C++ object.
  + And we can ask the object to perform operations for us.
* Vectors can grow and shrink in size at runtime, so it’s a perfect choice to model our student problem.
  + That’s the beauty of vectors and of the other classes in the STL library. The vector will take care of allocating or de-allocating space, adding new values to that space and so forth.
* Vectors also provide syntax and semantics similar to arrays.
* Like arrays, vector data is stored contiguously in memory.
* Also, after learning arrays, understanding vectors will be much easier.
* In addition to array-like syntax, we can also use lots of really powerful functions like sort, reverse, erase, find and more.
* Vectors also give us the ability to use methods that provide bounds checking if we wish.
  + If we use the subscript operator (square brackets) to access vector elements instead of the STL library OOP syntax (.at), then vectors will provide no bounds checking. This provides the same behaviour as arrays.
    - However, vectors provide a rich set of functions that provide type checking.
* Unlike arrays, when you declare a vector, the vector elements will automatically be initialized to 0 unless you specify otherwise.
* Let’s see how we can declare vectors.

#### Coding Info

##### Declaring Vectors

* There are several ways to declare vectors. The syntax for declaring a vector changes slightly from that of declaring an array because vectors are objects.
* First, we must include the vector library:

#include <vector>

* Also, the vector type is part of the standard library. So we must either use the namespace or use standard and the scope resolution operator.

using namespace std;

* Now we can create a vector of any type we want, just like we did with arrays.
* Since the vectors are object-oriented template class, we must include the type of the elements of the vector inside angle brackets.

vector <char> vowels;

vector <int> test\_scores;

* Both these examples create an empty vector that contains no elements.
  + Let’s look at a few other ways to declare vectors.

vector <char> vowels (5);

vector <int> test\_scores (10);

* In the first example, we declare vowels to be a vector containing 5 characters, test\_scores vector containing 10 integers.
* In this case, we’re not providing an initializer list, we’ll do it later. Instead, we’re using a constructor initialization syntax which provides information inside parantheses. This tells the compiler that we want 5 characters.
* Like in the second example, unlike arrays, these 10 integers will be automatically set to 0.
  + We don’t have to initialize all elements to 0, like we did in arrays.

##### Initializing Values to Declared Vectors

* As a reminder, unlike arrays, declared vectors’ values are automatically initialized to 0. We don’t have to do it manually.
* We can also use initializer lists as we did with arrays to initialize vector elements.

    vector <char> vowels\_initialized {'a', 'e', 'i', 'o', 'u'};

    vector <int> test\_scores\_initialized {100, 98, 89, 85, 93};

    vector <double> hi\_temperatures\_initialized (365, 24.3);

* In first example, we declare vowels to be a vector containing 5 characters, these characters are ‘a’, ‘e’, ‘I’, ‘o’, ‘u’.
  + Second example is also similar but with integers
* In the third example, NOTICE the paranteses instead of spring brackets. We used a constructor type, entered 365 as the vector size parameter, then 80.0 as the fill parameter to make all of the 365 elements to 80.0.
* There are also many other ways to declare vectors. You can declare a vector to be a copy of an array or a copy of another vector.

##### Accessing Vector Elements

* Accessing elements of vectors is same as accessing the elements of arrays.

    cout << hi\_temperatures\_initialized[3] << endl;

    cout << hi\_temperatures\_initialized.at(3) << endl;

##### Changing the Contents of Vector Elements

cin >> hi\_temperatures\_initialized.at(4);

hi\_temperatures\_initialized.at(5) = 20.9;

hi\_temperatures\_initialized[5] = 20.9;

##### When do they change size dynamically (Pushback Method)?

* The vector has a method called pushback, that adds new element to the end of the vector.
  + Remember, all vector elements must be of the same type, so add the element of the same type.
* vector <int> test\_scores\_dynamic {100, 95, 99}; // size is 3 for now
* test\_scores\_dynamic.push\_back(80); // 100, 95, 99, 80 (size = 4)
* test\_scores\_dynamic.push\_back(90); // 100, 95, 99, 80, 90 (size = 5)
* If this were an array and we wanted to add another test score, we’d be stuck since arrays are fixed in size.
  + However with vectors, we can use the pushback method and provide the integer you want to add within the parantheses.
  + That’s the beauty of vectors and of the other classes in the STL library. The vector will take care of allocating or de-allocating space, adding new values to that space and so forth.
  + You can concentrate on solving your problem.

##### What happens if you are out of bounds?

* If you go out of bounds, and you’re using a method that does bounds checking, C++ will throw an exception.
  + That’s the jargon used to say that an exceptional condition has been encountered and the program can’t continue.

vector <int> test\_scores\_boundary {100,95};

test\_scores\_boundary.at(2) = 97;

Exception has occurred: Unknown signal: terminate called after throwing an instance of 'std::out\_of\_range'

what(): **vector**::\_M\_**range\_check**: \_\_n (**which is 2**) **>=** **this->size() (which is 2)**

* But if we were used this:

    test\_scores\_boundary[2] = 97;

* It doesn’t generate any errors and may continue to execute the program or may crash, depending on the content at that memory location.

##### Removing Selected Element from Vector (Erase Function)

* .erase function’s input parameters are a little bit complicated because it expects an iterator input but here’s a example usage:

                int element\_to\_be\_removed{};

                int valid\_input{};

                cout << "[ ";

                for (int item : list\_int)

                {

                    cout << item << " ";

                }

                cout << "]" << endl;

                while (valid\_input == 0)

                {

                    cout << "Which element you want to remove from above?: ";

                    cin >> element\_to\_be\_removed;

                    for (size\_t i{}; i < list\_int.size(); i++)

                    {

                        if (element\_to\_be\_removed == list\_int.at(i))

                        {

                            valid\_input = 1;

                        }

                    }

                    if (valid\_input == 0)

                    {

                        cout << "Invalid input, please enter again: " << endl;

                    }

                }

                auto rem\_index = std::find(list\_int.begin(), list\_int.end(), element\_to\_be\_removed);

                list\_int.erase(rem\_index);

##### Multidimensional Vectors

    vector<vector<int>> movie\_ratings

    {

        {1, 2, 3, 4},

        {5, 6, 7, 8},

        {9, 10, 11, 12}

    };

A screenshot of a computer program

Description automatically generated

    cout << "Here's the rate of 1st movie reviewed by 1st reviewer: " << movie\_ratings[0][0] << endl;

Output: Here's the rate of 1st movie reviewed by 1st reviewer: 1

* or with out-of-bounds checking and more modern way:

        cout << "Here's the rate of 1st movie reviewed by 1st reviewer: " << movie\_ratings.at(0).at(0) << endl;

##### Why sizeof operator doesn’t work in vectors and what to use instead? (.size)

Case:

    vector <int> vec {10, 20, 30, 40 ,50};

    vec.at(0) = 100;

    cout << "Size calc: " << sizeof(vec)/sizeof(int) << endl; // This outputted 6 while i was expecting 5

    vec.at((sizeof(vec)/sizeof(int))-2) = 1000; // WHY DID WE USE -2 FOR THE LAST ELEMENT? SIZEOF COUNTS THE VANILLA ARRAY AS 6 ??

* In C++, sizeof behaves differently when used with vectors compared to arrays. Let's break this down:
* **sizeof and Vectors**: When you use sizeof on a vector, it doesn't give you the number of elements. Instead, it gives you the size in bytes of the entire std::vector object itself. This includes internal metadata like the capacity, size, and a pointer to the actual data, not just the elements in the vector.
  + In your case, sizeof(vec) is giving you the size of the vector object in memory, which includes this extra metadata, not just the elements stored in the vector.
  + **For arrays**: sizeof(array)/sizeof(type) works because arrays store only elements and no extra metadata.
  + **For vectors**: Use .size() to get the number of elements, since sizeof counts internal data, not just elements.
* **Why sizeof(vec)/sizeof(int) gives 6?**: sizeof(vec) is not giving you the size of the integer elements; it’s giving you the size of the vector object, which happens to be larger than just the size of the elements. When you divide by sizeof(int), you’re performing an incorrect calculation because sizeof(vec) includes more than just the elements themselves.
* **Correct Way to Access Vector Size**: To get the number of elements in a vector, use the .size() method. For example:

vec.at(vec.size() - 1) = 1000; // Access the last element

* This is the correct way to access elements from a vector, because .size() gives you the actual number of elements stored.

## Assignment Operator

* Because of C++ being statically typed, which means that a lot of errors are caught by the compiler at the compiling stage.
* This is very different from other languages like Python, Ruby and so forth, where they do all this type checking at runtime.
* C++ does it at compile time. So when the program is running you’re guaranteed that this is going to be correct because it’s already done a lot of this checking for you.
* When making an assignment or math operation, compiler checks that “does it make sense for me to assign whatever value was in here to here”.
* It determines that by looking at the variable types.

## Arithmetic Operations

### Modulo/Remainder (%) Operator

* Modulo or remainder operator (%) only works with integers. Can’t work with floats.

### Dividing two integers to get a float result

* While dividing two integers, you’ll get an integer result even while you were expecting a float result. For example,

int num1 {100};

int num2 {200};

int result = num1 / num2;

cout << "Result of 100/200 as they're both ints: " << result << endl;

* As a result, you’ll get 100/200 = 0.
* If you were did use floats in just one of these divided numbers, and declare the result as float, the result becomes normal:

    float num1{100};

    int num2{200};

    float result = num1 / num2;

    cout << "Result of 100/200 normal: " << result << endl;

* One of the operands and the result must be float, to get a float result.

### Processing Priority (PEMDAS) Rule in Math

1. P : Parantheses
2. E : Exponents
3. M : Multiplication & D : Division
4. A : Addition & S : Substraction

* If same prioritized elements present, go from left to right.

### Increment – Decrement Operators

* If you use increment or decrement operator on a float, it’ll increment or decrement the variable by 1.00. Decimal area doesn’t get effected.
* There are two variants to this operator.
  + Prefix notation (++num)
  + Postfix notation (num++)
* WARNING 1: Don’t overuse this operator.
* WARNING 2: Never use it twice for the same variable in the same statement. Because the behaviour is undefined and you’ll get a very unexpected result.

    int cnt {10};

    cout << "cnt original: " << cnt << endl;

    cnt++;

    cout << "cnt++: " << cnt << endl;

    ++cnt;

    cout << "++cnt: " << cnt << endl;

    // They both end to the same result but if you do these operations inside the cout statements,

    // ++cnt results quicker, cnt++ results latter. But in this example, it doesn't matter

* Let’s review what’s the difference between cnt++ and ++cnt.

#### Pre-Increment

    int cnt {10};

    int result {0};

    cout << "Counter at the beginning: " << cnt << endl;

    result = ++cnt; // First, pre-increment cnt, then assign cnt to result, RESULT IS INCREMENTED

    cout << "Counter: " << cnt << endl; // Resulted 11

    cout << "Result: " << result << endl; // Resulted 11

#### Post-Increment

    int cnt {10};

    int result {0};

    cout << "Counter at the beginning: " << cnt << endl;

    result = cnt++; // First, assign cnt to result, then inrement cnt, RESULT ISN'T INCREMENTED

    cout << "Counter: " << cnt << endl; // Resulted 11

    cout << "Result: " << result << endl; // Resulted 10

* As you can see the clear difference, the incrementing & assigning timings are different and it’ll be critical in some applications, especially in pointers.

## Mixed Type Expressions and Conversions

### Mixed Type Expressions

* This is when you have an expression where the operands are of different types. For example, a+b where a is an integer, b is a double.
* C++ is very consistent with it’s application of an operator to operands.
* The operants must be of the same type.
* It’s very important to understand the rules that C++ uses to ensure that the types are the same since the results of the calculation could be different depending on which operand type is changed.
* C++ will try to convert one of the operands so it matches the other.
  + In many cases, this happens automatically.
* If an automatic conversion (coercion) is not possible, then a compiler error will occur.
  + We saw an example of this in the assignment operator video when we tried to assign a string to an integer.
* In order to understand how these conversions happen, we need to understand higher versus lower types. The idea is simple:
  + Lower types can hold smaller values
  + Higher types can hold larger values
  + A long double is of higher type than a long,
  + A long is of higher type than an int

The hierarchic order from higher to lower is:

* long double -> double -> float -> unsigned long -> long -> unsigned int -> int
* The idea is we can typically convert from a lower type to a larger type automatically since the lower types value will fit into the higher types value but the opposite may not be true.
* “Short” and “character” types are always converted to “integers”.

Now, let’s learn the terminology

* **Type conversion** **(coercion):** Conversion of one operand to another type.
  + Sometimes this happens automatically and sometimes we do it ourselves in code.
* **Promotion**: When we convert a lower type to a higher type.
  + An example would be when we add an integer and a double.
  + In this case, we promote the integer to a double and then do the calculation.
* **Demotion**: When we convert a higher type to a lower type.
  + Suppose we want to store 12.5 into an integer variable.
  + It won’t fit. So a demotion happens, and the decimal part of 12.5 is truncated, and we’re left just with the integer 12.
* Now let’s see a few examples.

1. 2 \* 5.2
   1. (lower op higher)
   2. Lower (2) is promoted to higher (2.0)
   3. In order to perform the multiplication, the compiler will convert the lower type to the higher type and then perform the multiplication.
2. int num {0};

num = 100.2;

* 1. (lower = higher)
  2. The higher (100.2 (float)) is demoted to a lower (0 (int))
  3. It potentially loses information (decimal info) when demoting. Many compilers will warn you about the possible loss of precision but not all do.
* As mentioned, the C++ compiler will try to do automatic coercion when it can.
* However, as programmers we can explicitly tell the compiler to cast a specific type if we wish. Let’s see how:

### Explicit Type Casting – static\_cast<type>

* So let’s talk about how we can tell the compiler to coerce or cast one type to another. Let’s walk through this example:

int total\_amount {100};

int total\_number {8};

double average {0.0};

average = total\_amount / total\_number;

cout << average << endl; // displays 12

average = static\_cast<double> (total\_amount) / total\_number;

cout << average << endl; // displays 12.5 after casting

cout << "New type of total\_amount: " << typeid(total\_amount).name() << " (it's still an integer, just casted earlier)" << endl;

* We could have changed the type of total amount to double without doing the casting, but we’re modeling a running total of integers. So an integer type is more appropriate than double.
* The solution for us to tell the compiler to cast/perform a coercion of total amount from integer to double.
  + Now one of the operands (total amount) is a double FOR ONLY THAT SENTENCE, so the compiler will automatically convert the total number to a double and do double division, which is exactly what we want.
* Here’s another useful example of type casting:

const int num\_count {3};

int num1 {}, num2 {}, num3 {};

int total {};

cout << "Enter " << num\_count << " numbers with a space between them: ";

cin >> num1 >> num2 >> num3;

total = num1 + num2 + num3;

average = total / static\_cast<double>(num\_count);

//average = total / (double)num\_count; // Old C-type style, not recommended anymore

cout << num1 << " + " << num2 << " + " << num3 << " = " << total << endl;

cout << "Average of these " << num\_count << " numbers is " << average << "." << endl;

Output:

Enter 3 numbers with a space between them: 100 20 8

100 + 20 + 8 = 128

Average of these 3 numbers is 42.6667.

* The commented type casting is the old C type cast and we don't want to use it in Modern C++.
  + C++ type cast is a little bit more restrictive than C cast. C cast just assumes that whatever the total is, is going to be converted to a double. The static cast double checks to make sure that it can be converted to a double.

## Testing for Equality

### Comparing two doubles too close to each other, then fails

* When comparing two doubles very close to each other like this below:
  + 12.0 and 11.999999999999999

    double double1{}, double2{};

    cout << "Enter two doubles separated by a space: ";

    cin >> double1 >> double2;

    bool equal\_result = (double1==double2);

    bool not\_equal\_result = (double1!=double2);

    cout << "Comparison result (equals): " << equal\_result << endl;

    cout << "Comparison result (not equals): " << not\_equal\_result << endl;

Example output will be:

Enter two doubles separated by a space: 12.0 11.9999999999999999

Comparison result (equals): 1

Comparison result (not equals): 0

* Now to us, those are not the same number.
* But remember, the way the compiler stores floating point numbers, it stores approximations. Computer thinks these are equal.
  + Because the representation that it’s using to store them behind the scenes is equal.
* If you’re dealing with applications that need this kind of precision, scientific application, medical applications, safety critical type applications, you would not use these built-in doubles.
  + Instead, we would use specific specialized libraries that really help us deal with this.

### Comparing one integer to one double (mixed-mode comparison)

    int integer1{};

    double double3{};

    cout << "Enter one integer, then one double, separated by a space: ";

    cin >> integer1 >> double3;

    bool equal\_result = (integer1==double3); // Here, the integer is automatically promoted to double, only then compared

    bool not\_equal\_result = (integer1!=double3); // Here, the integer is automatically promoted to double, only then compared

    cout << "Comparison result (equals): " << equal\_result << endl;

    cout << "Comparison result (not equals): " << not\_equal\_result << endl;

Example output:

Enter one integer, then one double, separated by a space: 10 10.0

Comparison result (equals): 1

Comparison result (not equals): 0

* At first, computer doesn’t want to compare different types of data (integer-double). Those comparison elements have to be the same type.
  + So 10 will be promoted to a 10.0.
  + Then computer does the (10.0 == 10.0) comparison in this example.
  + Finds it true as expected.
* The issue explained in previous topic also happens here if we try to compare too close values like 10 and 9.99999999999999999999. Computer converts 10 to 10.0 and decides it’s same as 9.9999999999999999999, however it’s not.

## Relational Operators

### Three-way Comparison Operator (<=>)

* In C++20, there’s a really neat three-way comparison operator (<=>).
  + This operator compares two expressions,
    - If they’re equal, results 0.
    - If the left side is greater than the right side, results <0.
    - If the right side is greater than the left side, resutls >0.

## Logical Operators

### Precedence (Öncelik)

* Just like we had to understand precedence with the mathematical operators, we need to understand precedence for the logical operators.
* The precedence order is like below
  + not > and > or
* We can use parantheses to achieve exactly the behaviour we’re looking for, like we did in mathematical expressions.
* Not is an unary operator, and & or are binary operators.

### Short-Circuit Evaluation

* When evaluating a logical expression, C++ stops further evaluating as soon as it knows the result.
* For example, suppose we have an expression like the first one:
  + expr1 && expr2 && expr3
  + In and statement, if one of the elements is false, the whole statement will be false. So if expr1 is false, C++ will stop there, results as false and won’t continue evaluating expr2 and expr3 because they won’t matter.
  + Short circuit evaluation is also used in may other programming languages.
* In this example:
  + expr1 || expr2 || expr3
  + If any of the expressions is true, then the entire logical expression will be true. So as soon as C++ finds the true expression, it won’t continue evaluating other expressions to save time and resources and then make the whole statement as true.

## Compound Assignment

A table with text and symbols

Description automatically generated

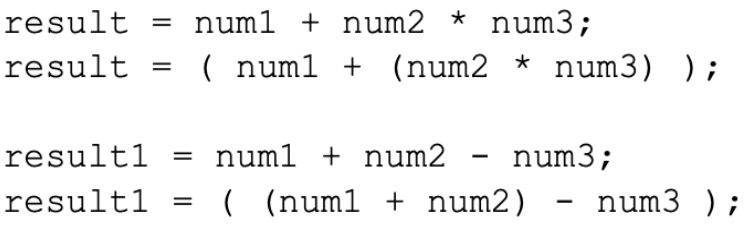
* The best tip when writing these operators is, think of the right side as being inside parantheses and you’ll always have it right.

## Operator Precedence

A screenshot of a computer

Description automatically generated

* C++ has well defined operator precedence and associativity rules.
* Above is a table showing the precedence and associativity of some of C++ operators.
  + The operators on higher rows have higher precedence than operators on lower rows.
  + Operators on the same row has same precedence.
  + This table is not complete, it only shows a subset of C++ operators.
* The second column is the associativity (çağrışımsallık?) column. This is the part that many times confuses students and even C++ programmers. Notice that associativity is left to right or right to left. Let’s see what it means:
* What’s associativity all about? Suppose you have an expression with two adjacent operators and these operators are different. Like we see in the first example.
  + expr1 **op1** expr2 **op2** expr3 // precedence
  + We can look for operators in the precedence chart, and if one of the operators has higher precedence than the other, then that’s the operator that’s applied first.
  + But suppose the two operators are the same or they’re different but they have the same level of precedence, now precedence doesn’t really help us.
    - In this case, we determine how the operators are applied by using their associativity.
    - We use associativity rules when adjacent operators have the same precedence.
    - For example
  + expr1 **op1** expr2 **op1** expr3 // asociativity
    - If op1 associates left to right, then we apply the operator to expression1 and expression2 first, then apply result to expression3.
    - If op1 associates right to left, then we apply the operator to expression3 and expression2 first, then apply result to expression1.
  + Of course you can always use parantheses to remove any doubts and be sure that your result is what you expect.
* Using parantheses is good practice with complex expressions.
* In this example below



* We know that in the first example, the precedence order follows as multiplication -> addition -> assignment.
* In the second example, both addition’s and subtraction’s precedence level is same. Now we have to look at it’s associativity, which is left to right.

## If-Else Statements

### Semi-Colon after if statement

    int temperature {70};

    if (temperature < 50);

        cout << "It's cold!" << endl;

In above example, semi-colon at the end of the first if-statement means do nothing if the condition is true.

Meaning, in any case, it’s gonna print “It’s cold” because this line isn’t inside of the if statement anymore, because of the semi-colon. So it acts like a comment line.

## Switch-Case Statements

### Main Overview

A screen shot of a computer code

Description automatically generated

* The value of the control expression will be compared to other case values.
  + These case statements must also evaluate to an integral (consists many subtypes of characters and integers) or enumeration type and must be known at compile time.
    - That means case expressions must be constants or literals.
* The case expressions must be known at compile time.
* When the value of the control expression matches, the case expression then the code after the colon is executed until it hits a break statement.
* Break statements are optional, but best practice is to include break statements for every case statement (unless you’ve got a really good reason not to).

### Performance comparison with if statements

* For just a few items, the difference is small. If you have many items you should definitely use a switch.
* If a switch contains more than five items, it's implemented using a lookup table or a hash list. This means that all items get the same access time, compared to a list of if:s where the last item takes much more time to reach as it has to evaluate every previous condition first.

### Enumeration Input with Switch Case

* Using enumeration variable in the switch statement is a very common use case and a great way to execute code depending on it’s value

    enum Color

    {

        red, green, blue

    };

    Color screen\_color {green};

    switch (screen\_color)

    {

        case red:   cout << "red";      break;

        case green: cout << "green";    break;

        case blue:  cout << "blue";     break;

        default:    cout << "should never execute";

    }

* In above example, we’re supplying a default case but this should never execute unless we add another enumeration constant to our enumeration and forget to include a case for it.

#### Benefits

* Using enumerations in switch-case mechanisms has the main benefit of achieving compiler warnings if
  + There are unhandled cases OR
  + Default case is not provided
    - These warnings be like: Enumeration value ‘x’ is not handled in switch.
* Compiler understands that there are more enumeration constants but some of them are not handled in switch cases, maybe somehow forgotten.
* A lot of other programming languages won’t compile in a situation like this and will give an error and they force you to handle all the types.
* To fix this, you could create two more types for missing cases or declare a default case, which’ll catch everything else.

### One Statement for Multiple Cases

    switch(letter\_grade)

    {

        case 'a':

        case 'A':

            cout << "You need a 90 or above for A! << endl;

            break;

        case 'b':

        case 'B':

            cout << "You need 80-89 for a B" << endl;

            break;

        default:

            cout << "You shall not pass!" << endl;

    }

### If-Else Statement inside Switch-Case

        case 'f':

        case 'F':

            {

                char confirm {};

                cout << "Are you sure (Y/N)?" << endl;

                cin >> confirm;

                if (confirm == 'y' || confirm == 'Y')

                {

                    cout << "Then, you shall not pass !!" << endl;

                }

                else if (confirm == 'n' || confirm == 'N')

                {

                    cout << "I knew you wouldn't get an F." << endl;

                }

                else

                {

                    cout << "Invalid input" << endl;

                }

                break;

            }

## Conditional Operator

* This is a really neat and very useful operator.

(cond\_expr) ? expr1 : expr2;

* Conditional operator is a ternary operator, which means that it operates on 3 operands.
  + The first operand is the conditional expression, which is usually in parantheses.
  + The conditional expression is evaluated first and it must evaluate to a Boolean or true/false value.
  + Then we follow the conditional expression with the question mark part of the conditional operator, followed by expression 1, then the colon part, followed by expression 2.
* If the conditional expression is true,
  + The value of expression 1 is returned.
* If the conditional expression is false
  + The value of expression 2 is returned.
* You can see it’s like an if-else construct in a single expression.
* Typically, it’s used inside a loop as we’re looping and we want to do something every fifth time, tenth time, etc.
  + Let’s say every fifth time, you want to print a new line or something like that.
  + Then this makes a lot of sense to do it inside an output statement like this.
* While this operator is very handy, it’s also very easy to abuse it.
  + Best practice is to NEVER NEST A CONDITIONAL OPERATOR.
  + This leads to if-else-if logic that quickly becomes unreadable and difficult to manage.

Some independent examples are below:

    int a{10}, b{20};

    int score {92};

    int result {};

    result = (a > b) ? a : b;

    result = (a < b) ? (b-a) : (a-b);

    result = (b != 0) ? (a/b) : 0; // avoiding divide by 0

    cout << ((score > 90) ? "Excellent" : "Good ");

Here’s another useful string example which decides if a number is odd or even:

    int num {};

    cout << "Enter an integer: ";

    cin >> num;

    cout << num << " is " << ( (num%2 == 0) ? ("even") : ("odd")) << "." << endl;

Another example that compares two integers and displays the largest and smallest

    int num1 {}, num2 {};

    cout << "Enter 2 numbers separated by a space: ";

    cin >> num1 >> num2;

    if (num1 != num2)

    {

        cout << "Largest: " << ((num1>num2) ? (num1) : (num2)) << endl;

        cout << "Smallest: " << ((num1<num2) ? (num1) : (num2)) << endl;

    }

    else

    {

        cout << "Numbers are same" << endl;

    }

## Looping

* C++ has 3 main looping structures
  + for loop
    - Also, range-based for loop,
      * Used when one iteration needed for each element in a range or collection.
      * Very handy and great for iterating through range of items or elements of a collection (arrays, vectors, etc.).
  + while loop
  + do-while loop

### For Loop

#### Basic Tips

* i++ and ++i mean same thing here since the increment happens on it’s own and it’s not in another expression.
* C++ for loops are very efficient and in their basic form, they are very easy to read and modify.
* Initialization style is also must be used when declaring I inside the for loop like below:
* The looping variable i is only visible inside the for loop. Once the loop is finished, i is no longer visible and reachable.
  + This is a great feature since you know that I is totally under your control within the loop and won’t have any strange values coming into the loop or exiting the loop.

#### Multiple criterias and multiple operations in a single for loop

* C++ has an operator called the comma operator. That isn’t used very often in C++. But sometimes, you see it used in loops to initialize loop variables. The comma operator allows you to separate expressions with a comma and both expressions will execute.
* Note that associativity is right to left, and the result of the comma operator ris the leftmost expression.

    for (int i {1}, j {10}; (i<=5 || j>=7); i++, j--)

    {

        cout << i << " \* " << j << " : " << (i\*j) << endl;

    }

Output:

1 \* 10 : 10

2 \* 9 : 18

3 \* 8 : 24

4 \* 7 : 28

5 \* 6 : 30

#### Using doubles in loop control

* Doubles can be used in for loops as well. And the increment-decrement steps is based on the user, like below:

    for (double i {0.00}; i<=1.70; i+=0.2)

    {

        cout << "i: " << i << endl;

    }

Output:

i: 0

i: 0.2

i: 0.4

i: 0.6

i: 0.8

i: 1

i: 1.2

i: 1.4

i: 1.6

#### Endless for loop

* You an have a for loop with just the semicolons in the parantheses. Notice the one sentence operation without curly brackets. Only the first sentence after for loop operator will be executed within a loop because of lackness of curly brackets.

    for (;;)

        cout << "Endless loop" << endl;

#### Range based for loop

* Range based for loop was added in C++11.
* This loop really makes C++ feel like a modern programming language.
* The idea with the range based for loop is to loop through a collection of elements and be able to easily access each element without having to worry about:
  + the length of the collection
  + incrementing or decrementing looping variables
  + subscripting indexes

A close-up of a code

Description automatically generated

* The syntax is very simple and elegant.
  + Inside the parantheses, we provide the type and name for the variable we want to use in the loop body.
    - This variable will be bound to each element of the collection.
    - So it should be of the same type as the collection elements.
  + Then we provide a colon and the collection or collection name.
* Now when we access the variable name in the body of the loop, it’ll have a specific element in the collection.

As an example:

    int scores [] {100, 90, 97};

    for (int single\_score : scores)

    {

        cout << single\_score << endl;

    }

Output:

100

90

97

##### “Auto” keyword

* We actually don’t have to explicitly provide the type of the variable.
* Instead, we can use “auto” keyword for this.
  + “Auto” tells the C++ compiler to deduce the type itself, based on the declarations.
* So in this case below, compiler sees that you’re using a collection of scores in a loop. So it looks at scores and sees that it’s an array of integers, so it uses an integer for the score variable.

    double scores [] {100.6, 90.3, 97.2};

    for (auto single\_score : scores)

    {

        cout << single\_score << endl;

    }

Output:

100.6

90.3

97.2

* In this case, using auto versus int doesn’t really buy us much, but C++ can have very complex collections. And sometimes defining the type of a collection variable can be quite long and tricky, “auto” makes it simple.

Another example using auto:

    vector <double> temps {87.2, 77.1, 80.0, 72.5};

    double average\_temp {}, running\_sum{};

    for (auto temp: temps)

    {

        running\_sum += temp;

    }

    average\_temp = running\_sum / temps.size();

    cout << "Average temp: " << average\_temp << endl;

Output: Average temp: 79.2

##### Providing collection right in the loop

    double average\_temp\_2 {};

    double running\_sum\_2 {};

    int size\_2 {};

    for (auto temp: {60.2, 80.1, 90.0, 78.2})

    {

        running\_sum\_2 += temp;

        size\_2++;

    }

    average\_temp\_2 = running\_sum\_2 / size\_2;

    cout << "Average temp 2: " << average\_temp\_2 << endl;

* The only downside is that you have to calculate the size as you go.

##### Using it on a string

    for (auto c: "Kemal")

    {

        cout << c << endl;

    }

Output:

K

e

m

a

l

Another example, space remover

    for (auto c: "Kemal Daysal")

    {

        if (c != ' ')

        {

            cout << c;

        }

    }

Output: KemalDaysal

## Do While

A common do-while loop usage is like below

    int number {};

    do {

        cout << "Enter a number between 1 and 5: ";

        cin >> number;

    } while (number < 1 || number > 5);

    cout << "Valid input, thanks." << endl;

Output:

Enter a number between 1 and 5: 7

Enter a number between 1 and 5: 5

Valid input, thanks.

## Continue and Break Statements

* Continue and break statements can be used within ALL of C++ loop constructs to provide more explicit control over the looping behavior.
* They are very easy to use but don’t overuse them.
  + If you have a loop that has 10 breaks and 12 continues in it’s body, that’s a very complex piece of code to try to understand.
  + The less ways that you can come into a loop and out of the loop, better we can understand and debug it.

### Continue

* When the continue statement is executed in the loop,
  + “No further statements in the body of the loop” are executed and
  + Control immediately goes directly to the beginning of the loop for next iteration.
* You can think of this as skip processing in the rest of this iteration and go to the beginning of the loop for next iteration.
* In case of while and for loops, just after continue statement is executed, the condition will immediately be tested again.

### Break

* When the break statement is executed in the loop,
  + “No further statements” in the body are executed and the loop is terminated.
  + So code continues to execute the first statement after the loop construct.

### Examples

#### Collection checker with exit and skip conditions

    // Write a collection checker and element writer program that exits if it sees -99 and skips -1s

    vector <int> arr {1, 2, -1, 3, -1, -99, 7, 8, 10};

    for (int element: arr)

    {

        if (element == -1)

        {

            continue;

        }

        else if (element == -99)

        {

            break;

        }

        else

        {

            cout << element << endl;

        }

    }

Output:

1

2

3

## Infinite Loops

### Overview

* Usually, infinite loops are mistakes by the programmer.
* But sometimes infinite loops are created on purpose. And programmers use break statements at strategic places to break out of the loop.
* There are cases where this makes sense and could be justified.
  + However, writing an infinite loop and using break statements to terminate the loop is usually bad practice and should be rewritten so the loop condition expression is descriptive fo what the loop is doing.
* Infinite event loops are commonly used in event-driven programs such as those you find on mobile devices and embedded systems.
  + In these programs, the program loops forever for example listening for mouse clicks, movements, touches and so forth and reacting to them.
  + This continues as long as the program is running.
* Another example of an endless loop is an operating system. An OS is constantly looping handling input/output, handling resources and so forth. It only shuts down when you shut down your computer.

### Examples

#### Infinite for loop

    for (;;)

        cout << "This'll print forever" << endl;

* Remember that all 3 expressions in the for loop are optional. So if you omit all 3 expressions, you get just 2 semicolons. By definition in C++, that’s an infinite loop.

#### Infinite while loop

   while (true)

    cout << "This'll print forever" << endl;

* Below example won’t work and the sentence won’t be written because the condition for while loop to execute is the control statement must result as 1.

while (false)

cout << "This'll print forever" << endl;

#### Infinite do-while loop

    do {

        cout << "This'll print forever" << endl;

    } while (true);

#### Quit Event Checker

    while (true)

    {

        char response{};

        cout << "Do you want to loop again? (Y/N): ";

        cin >> response;

        if (response == 'n' || response == 'N')

            break;

    }

## Nested Loops

* Nested loops have many uses but they’re especially useful for multi-dimensional structures, like 2D, 3D arrays and 2D, 3D vectors.

### Examples

#### Multiplication table from 1 to 10 (10x10)

    for (size\_t num1{1}; num1 <= 10; num1++)

    {

        for (size\_t num2{1}; num2 <= 10; num2++)

        {

            cout << num1 << " \* " << num2 << " = " << num1\*num2 << endl;

        }

        cout << "----------------" << endl;

    }

Output:

1 \* 1 = 1

1 \* 2 = 2

1 \* 3 = 3

1 \* 4 = 4

1 \* 5 = 5

1 \* 6 = 6

1 \* 7 = 7

1 \* 8 = 8

1 \* 9 = 9

1 \* 10 = 10

----------------

2 \* 1 = 2

2 \* 2 = 4

2 \* 3 = 6

2 \* 4 = 8

2 \* 5 = 10

2 \* 6 = 12

2 \* 7 = 14

2 \* 8 = 16

2 \* 9 = 18

2 \* 10 = 20

And so forth…

#### Filling and Displaying 2D Arrays

    int arr[4][4] {};

    int filler {};

    for (size\_t row{0}; row < 4; row++)

    {

        for (size\_t col{0}; col < 4; col++)

        {

            arr[row][col] = filler;

            cout << arr[row][col] << " ";

            filler++;

        }

        cout << endl;

    }

Output:

0 1 2 3

4 5 6 7

8 9 10 11

12 13 14 15

#### Nested Loops with Vectors

    vector<vector<int>> vector\_2d{

        {1, 2, 3},

        {10, 20, 30, 40},

        {100, 200, 300, 400, 500}};

    for (auto row : vector\_2d)

    {

        for (auto col : row)

        {

            cout << col << " ";

        }

        cout << endl;

    }

Output:

1 2 3

10 20 30 40

100 200 300 400 500

* In above example, 4 loops are used within 4 loops.

#### Creating a dynamic vector and displaying it’s elements as a histogram

    size\_t item\_count{};

    cout << "How many items will you enter?: ";

    cin >> item\_count;

    vector<int> arr{};

    for (size\_t i{0}; i < item\_count; i++)

    {

        cout << "Enter item " << i << ": ";

        int single\_item{};

        cin >> single\_item;

        arr.push\_back(single\_item);

    }

    for (auto val : arr)

    {

        for (int i{}; i<val; i++)

        {

            cout << "-";

        }

        cout << endl;

    }

Output:

How many items will you enter?: 3

Enter item 0: 5

Enter item 1: 7

Enter item 2: 2

-----

-------

--

## Characters and Strings

* C++ supports 2 types of strings, which are C-style strings and C++ strings.
  + You should be using C++ strings in Modern C++. The reason for this’ll be discussed at the end of this topic.
* First, we’ll look at the Cctype library.
  + This is a library of a very simple but very useful functions that work with characters.
  + For example, we can test characters to see if they’re uppercase, lowercase, numeric, alphanumeric, punctuation and more.
  + We can also convert characters between lower and uppercase.
* After we learn about Cctype functions, we’ll talk about strings. Because a string is a sequence of characters. So far, we’ve only used string literals, which were represented as a sequence of characters inside double quotes.
* First, we’ll see what C-style strings are and how they represented in memory. Then we’ll see how to work with C-style strings using some of the string functions available in the C-string and C-standard lib libraries.
* Then we’ll learn about C++ strings.
  + C++ strings are objects, just like are vectors were.
  + They’re used using an object-oriented style of programming.
  + In this section, we’ll learn how to declare, initialize, assign, compare and use some of the other C++ string methods.
    - Methods are functions that work with objects.

### Character Functions

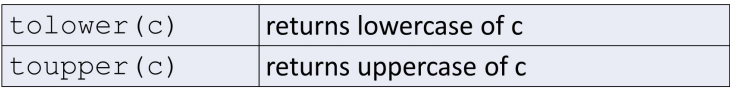
* cctype library includes very simple and useful functions that allow the testing of characters for various properties as well as the conversion of characters between upper and lower cases.
* In order to use these functions, cctype must be included.
* All functions expect a single character input.
* In the case of the testing functions, they evaluate to true or false.
* And the conversion functions return the converted character.

#### Functions – Character Tests

A table of black and white text

Description automatically generated

#### Functions – Character Conversions



* Both evaluate output of the desired upper or lowercase character.
* If these functions can’t perform the conversion, then they simply evaluate to the original character that was passed in.
* These functions are both very handy and extremely efficient.

### C-style Strings

* They’re simple, sequence of characters, stored contiguously in memory.
* They’re implemented as an array of characters, so you can access individual characters using the array subscript syntax that we’ve already learned.
* How do you know where the sequence of characters ends?
  + C-style strings use a sentinel valley that marks the end of the string.
  + The null character is used, which is equivalent to the integer 0.
    - So, C-style strings are often referred to as 0 or null terminated strings.

#### Declaring and Changing Strings

* We’ve been using C-style strings all along in this course. We’ve used them mainly in our output statements as string literals.
  + Recall that string literals are sequences of characters enclosed in double quotes.
  + These string literals are constants, so we can’t change them just like we can’t change integer literals.
* Even though we don’t explicitly provide a null character at the end of the string literal, C++ inserts one for us. That’s the way the end of the string is handled.
* Let’s see an example of a string literal and how it’s stored in memory.

A close-up of a box

Description automatically generated

* These characters are stored in a contiguous block of memory, thus they can be accessed as an array.
* Notice that “C++ is fun” has exactly 10 characters, but the compiler allocated 11 characters for the array because it needed space for the null character at the end.
* Literals are great and very useful, but we more often than not need to create string variables.
* In below example, notice that C++ compiler will allocate 6 characters for this array. And also the size of the array is fixed. So if we wanted to add a ‘y’ to “Frank” and create “Franky”, we couldn’t without having some potential problems since the new string wouldn’t be null terminated.
  + You won’t get a compiler error or even a warning because my\_name[5] is still within the array bounds.

A screenshot of a computer code

Description automatically generated

    char my\_name[] {"Frank"};

    cout << my\_name << endl;

    my\_name[5] = 'y';

    cout << my\_name << endl;

Output:

Frank

Franky��\_

* Notice that because we replaced null character with another literal character, it couldn’t end the Franky string and displays what comes after it until it reaches an undefined null character.
* In below example, we’re explicitly asking the compiler to allocate 8 chars for the character array and we’re initializing it to “Frank”. It’ll initialize the array, then fill the remaining spaces with the null characters (\0) for termination.
  + In this case, if we try to change 5th character to ‘y’, it’ll be fine because it still has 2 more termination (null) characters left.

A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

* In above example, we ask the compiler to allocate an array of 8 characters and we don’t initialize them. This could be very problematic because if you use this array as a string, all c-style string functions expect to find a null character. And here there may or may not be one. We really don’t know what the data in the array is.
* Suppose we wanted to display the string now. How do you think this would happen? How do you think c-style strings are displayed?
  + In this case, we start at the first element of the array and we iterate through the array. At every character, we check if it’s the null character. If it’s a null, we stop since we reached the end of the string. If it’s not null, we print the character and then we move to the next character.
* In this case, we don’t know what’s going to happen. We might see a null right away or we might not see a null for a very long time.
* In the meantime, I’m printing stuff to the screen. Some of the stuff may not even be displayable since some characters are control characters. You might think we can initialize c-style strings using the assignment operator, just like we have with other types, but this won’t work. If we try to assign the c-style literal “Frank” to my\_name, we get a compiler error because that’s not the way c-style strings work.
* Array names and literals evaluate to their location in memory, so we’re effectively assigning one location to another which is illegal.
  + Think of this as saying “assign 10 to 12”. It doesn’t make sense and the compiler won’t allow it.
  + In order to assign one string to another, we need to use a function called string copy (strcpy).
    - Strcpy copies the c-style literal “f r a n k” to the my\_name array.

#### Functions that work with c-style strings

* There are lots of c-style functions that C++ brings in from the C language.
* These functions are used to copy, concatenate, compare, search, get the length of a string and many more.
* They all rely on one common factor that the sequence of characters they’re working on is terminated with a null character.
  + If you’re using c-style strings, you must be VERY aware of this.
  + For example, suppose you have a c-style string and you want to determine it’s length. Let’s say we have “Frank”. Since no length information is stored with the string, the only way to determine the length is to start counting at the first array element and then increment until you see the null character. Then count the characters until the null termination.
    - But if this null character is missing, it’ll keep counting so the result will very likely be incorrect, even worse if you’re copying one string into another, and the string you’re copying isn’t null terminated, you’ll very likely exceed the bounds of the target string and very likely cause a program crash.
  + Although this sounds ominous, we have been programming in C and C++ using c-style strings for a very long time and we’ve gotten pretty good at understanding these issues.
    - But wouldn’t it be great if we didn’t have to worry about these issues at all.
      * That’s where C++ strings come in, we’ll talk about them later in this section.

#### Cstdlib Library

* C++ also has another library called c standard lib that contains functions that convert strings to other types like integers, floats, longs, etc.
  + For example, if I have a string with the digits 1 and 2, I can convert that to integer 12.
* All of the string functions including this works with null terminated strings. If you ‘re using c-style strings, you have to be very aware of that.

#### Examples and Common Mistakes

##### Proper way of assigning a value containing a whitspace to a string

* In below example, at the first code snippet, the full\_name string doesn’t get read properly since there’s a whitespace between first name and last name. cin function stops reading after it sees a whitespace character like a blank or space and stops there, just takes the first input.
* In order to avoid this, we’ll use cin.getline(place\_to\_be\_copied, size) function.

    cout << "Enter your full name: ";

    cin >> full\_name; // Won’t read last name because cin stops reading when it sees a whitespace character. It just takes the first name

    cout << "Your full name is " << full\_name << endl;

    cout << "Enter your full name: ";

    cin.getline(full\_name, 50);

    cout << "Your full name is " << full\_name << endl; // True output

### C++ Strings

* Std::string is a class in the C++ STL (Standard Template Library) library.
* <string> header must be included to use C++ strings.
* Strings are in namepace std.
* Stored contiguously in memory, like C style strings.
* Dynamic and can grow, shrink as needed at runtime (unlike C style strings, which are fixed in size).
* Work with the stream (input/output) insertion and extraction operators, just like most other types in C++.
* Provides a rich set of methods/functions that allow us to manipulate strings easily.
  + If you need to do something with the string, that functionality is already there for you without having to rewrite it from scratch.
* They also work with most of the operators that we’re used to for assigning, comparing and so forth (+, =, <, <=, >, >=, +=, ==, !=, []…).
  + Huge advantage over C-style strings since C-style strings don’t work well with these operators.
* For these advantages, even though C++ strings are preferred in most cases, sometimes you need to use C-style strings. Maybe you’re interfacing with a library that’s been optimized for C-style strings.
  + In this case, while you can still use C++ strings and take advantage of them, you can also easily convert the C++ strings into C-style strings and back again when needed.
* Like vectors, C++ strings are safer since they provide methods that can bounce check and allow you to find errors in your code so you can fix them before your proram goes into production.
* The C++ string class has a very rich set of very useful and efficient methods, too many methods to cover in detail in this course. Instructor encourages us to study the C++ string class since it’s going to be a class that you’ll use often, and it’s important that you know what provides, so you don’t reinvent the wheel when you need to solve the problem.

#### Declaring and initializing

* We need to include <string> and using namespace std.
* Unlike C-style strings, C++ strings are always initialized automatically.
  + No garbage chars and memory to have to worry about.

    string s1; // initialized as empty automatically

    string s2 {"Frank"}; // Frank

    string s3 {s2}; // Frank

    string s4 {"Frank", 3}; // Fra

    string s5 {s3, 0, 2}; // Fr

    string s6 {s3, 2, 4}; // ank

    string s7 (3, 'X'); // XXX // Notice the parantheses using constructor syntax

#### Assignment

* With C++ strings, we can use the assignment operator.
  + This feels much more natural than having to use the strcpy function like we would have to in C-style strings.
* The assigned variable will grow dynamically as needed.

    string s1;

    s1 = "C++ Rocks!";

    string s2;

    s2 = s1; // We didn't use strcpy like we did in C-style strings, this is much easier

    string s3 {"Frank"}; // Frank

    cout << "s3 original: " << s3 << " with size of: " << s3.length() << endl;

    s3[0] = 'C'; // Crank

    s3.at(3) = 'p'; // Crap(space)

    s3.at(4) = '\0'; // Crap

    cout << "s3 new: " << s3 << " with size of: " << s3.length() << endl; // Crap with size 5 again, why??

#### Concatenation

* We can use + operator to concatenate C++ strings.

    string part1 {"C++"};

    string part2 {"is a powerful"};

    string sentence;

    sentence = part1 + " " + part2 + " language";

    // C++ is a powerful language

    // A combination of C++ strings and C-style strings is okay,

    sentence = "C++" + " is powerful"; // ILLEGAL !! Won’t compile

    // But this one won't compile because we have 2 c-style literals

    // You can't concatenate c-style literals, it only works for C++ strings.

    // It should have had at least 1 C++ strings like this:

    sentence = part1 + "C++" + " is powerful";

s string s3 = “nice ” + “ cold” + s5 + “juice” // Compiler error ???

#### Accessing characters with [] and at() method

* Just like we did with vectors, we can use the same operators to access string elements.
* In this case, the elements of a string are characters. So we can use subscript operator as well as the .at method.
  + Remember, the .at method performs bounce checking. So if you go over bounds, you’ll get an exception which you can fix.

    string s1 {"Frank"};

    for (char c: s1)

        cout << c << " "; // Outputs F r a n k

    for (int c: s1)

        cout << c << " "; // Outputs 70 114 97 110 107 for ASCII integer equivalents

#### Comparing

* It couldn’t be easier. We use the same equality and relational operators that we’ve been using all along integers etc.
* We’re comparing 2 string objects, so they’ll be compared character by character, and their character values will be compared lexically.
  + So, ‘A’ is less than ‘Z’, ‘A’ is less than ‘a’. Because capital letters come before the lowercase letters in the ASCII table.
* We can’t use these operators on 2 c-style literals, but we can use them in the following hybrid cases:
  + If we have 2 C++ strings,
  + If we have 1 C++ string and a 1 C-style string literal
  + If we have 1 C++ string and a 1 C-style string variable

Now, let’s get to the examples:

    vector <bool> results;

    string s1 {"Apple"};

    string s2 {"Banana"};

    string s3 {"Kiwi"};

    string s4 {"apple"};

    string s5 {s1}; // Apple

    results.push\_back(s1 == s5); // 1

    results.push\_back(s1 == s2); // 0

    results.push\_back(s1 != s2); // 1

    results.push\_back(s1 < s2); // 1

    results.push\_back(s2 > s1); // 1

    results.push\_back(s4 < s5); // 0

    results.push\_back(s1 == "Apple"); // 1 // comparing C++ string and C-style string literal

    cout << "Results: " << endl;

    for (auto element: results)

    {

        cout << element << endl;

    }

#### Substrings – substr()

* Substring method extracts a substring from a C++ string.
  + It doesn’t change the original string.
  + It simply returns the substring and you could do whatever you want with it.
* Function prototype is like below:
  + object.substr(start\_index, length)

    string s1 {"This is a test"};

    cout << s1.substr(0, 4); // This

    cout << s1.substr(5,2); // is

    cout << s1.substr(10,5); // test

    cout << s1.substr(12,6); // st // doesn't display garbage letters in memory

* In the last example with 12,6, there are 4 more letters we wanted to subscript after the string ends with “st”. But no garbage letters are displayed after st,

#### Searching – find()

* C++ string has a very handy method named find.
* Find works with characters and strings.
* It expects a string or character and returns the index or position.
* Function prototype is like below:
  + object.find(search\_string)

    string s1 ("This is a test");

    cout << s1.find("This") << endl; // 0

    cout << s1.find("is") << endl; // 2

    cout << s1.find("test") << endl; // 10

    cout << s1.find('e') << endl;  // 11

    cout << s1.find("is", 4) << endl; // 5 // search starting index is included

    cout << s1.find("XX") << endl; // returns:npos, 18446744073709551615 // invalid target

* In the last example, we see what happens if the string or character we want to find isn’t just there.
  + In this case, the method returns an n-position (string::npos, which means no position information available.
  + You can check for this value in an if statement.
    - And if true, you know what you were searching for wasn’t there.
    - Very easy, very powerful
* There’s also an r-find method that starts searching in the opposite direction, from the end of the string to the beginning of the string.

    cout << s1.rfind("test") << endl; // 10 // search from the end but display it as normal

Here’s another useful example below:

    string word {"Bob"};    // "The secret word is Bob

    string sentence {};

    cout << "Enter your sentence: ";

    getline(cin, sentence); // input: This house was built by Bob.

    size\_t position = sentence.find("Bob");

    if (position != string::npos) // if it's found (output is not npos)

        cout << "Found " << word << " at position: " << position << endl; // found at pos 24

    else

        cout << "Sorry, " << word << " is not found :(" << endl;

Here’s another example about Substitution Cipher:

    const string alphabet{"abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'.,!-\_:;<>()"};

    const string cipher  {"ZYXWVUTSRQPONMLKJIHGFEDCBA<zyxwvutsrqponmlkjihgfedcba9876543210\*\_-:!;,.][}{"};

    string message{};

    cout << "Enter your secret message: ";

    getline(cin, message);

    // Encryption

    // Pseudo code of encryption:

    /\*

        1) Loop through elements of original message

        2) Check every letter of the original message while looping

        3) Using .find, find that letter's index on the alphabet table

        4) Using .swap (maybe), swap that letter with the same index on cipher table

    \*/

    string encrypted\_message {};

    bool encryption\_error{0};

    for (char letter: message)

    {

        size\_t index = alphabet.find(letter);

        if (index == string::npos)

        {

            cout << "Unknown alphabet-cipher element (" << letter << ") detected, quitting." << endl;

            encryption\_error = 1;

            break;

        }

        // cout << "Index " << i << " is: " << index << endl; // For debugging

        encrypted\_message += cipher.at(index);

    }

    if (!encryption\_error)

    {

        cout << "-Your message is encrypted like below:" << endl;

        cout << encrypted\_message << endl;

        // Decryption

        // Pseudo code of decryption:

        /\*

            1) Loop through elements of encrypted message

            2) Check every letter of the encrypted message while looping

            3) Using .find, find that letter's index on the cipher table

            4) Assign that indexed alphabetic letter into decrypted message's i. element

        \*/

        string decrypted\_message {};

        for (char c: encrypted\_message)

        {

            size\_t index = cipher.find(c);

            decrypted\_message += alphabet[index];

        }

        cout << "-Here's your decrypted message below:" << endl;

        cout << decrypted\_message << endl;

    }

Example I/O:

Enter your secret message: I'll meet you at Walter's house when the sun goes down for the last time.

Your message is encrypted like below:

r\*OO<NVVG<BLF<ZG<dZOGVI\*H<SLFHV<DSVM<GSV<HFM<TLVH<WLDM<ULI<GSV<OZHG<GRNV\_

Here's your decrypted message below:

I'll meet you at Walter's house when the sun goes down for the last time.

#### Removing characters – erase() and clear()

* We can also remove characters from a C++ string using the erase and clear methods.
* For erase method, you provide the starting index and how many characters to delete.
* Clear method deletes all the characters in the string so the string becomes the empty string.

    // prototype: returns the remaining string = object.erase(start\_index, length\_to\_delete)

    string s1 {"This is a test"};

    cout << s1.erase(0, 5) << endl; // remaining: is a test

    cout << s1.erase(5, 4) << endl; // remaining: is a

    s1.clear(); // empties string s1

#### Length Method

    string s1 {"Frank"};

    cout << s1.length() << endl; // 5

* This is so easy and something that was impossible to do with C-style strings since they don’t contain size information.

#### Compound Concatenation Assignment

    string s1 {"Frank"};

    s1 += " James";

    cout << s1.length() << endl; // 11

    cout << s1 << endl; // Frank James

* This is really handy and works very much the same way the compound assignment operators worked with integers, doubles and so forth.

#### Input >> and getline()

* C++ strings work great with input and output streams. As you’ve seen, inserting C++ string variables to an output stream like cout is pretty easy and works just like we’ve been doing all along.
* Extracting a C++ string from an input stream like cin also works the same way we expect.
  + However, there’s 1 issue that’s also true for c-style strings.
* The issue is, extractor operator stops input reading when it sees a whitespace. If we type “Hello there”, it only takes “Hello”.
  + In order to achieve what we desire, we can use getline function, which has couple of variants. You can see this in below example.
* Getline function stops reading at newlines (\n) as default, but you can configure where to stop it by providing an extra parameter (third).

    // prototype: getline(input\_source, where\_to\_store\_it, (optional) stop\_sign)

    string s1;

    cout << "Enter a string input: ";

    cin >> s1; // Suppose we typed "Hello there"

                // But it won't accept the words after space, because extractor stops at whitespace

    cout << s1 << endl; // We'll only see "Hello"

    getline(cin, s1); // Read entire keyboard input line until \n and store it at s1

    cout << s1 << endl;

    getline (cin, s1, 'x'); // input: Extra // stop reading at ‘x’

    cout << s1 << endl; // output: E

#### Insertion Operator - .insert

    string unformatted\_full\_name {"StephenHawking"};

    string first\_name {unformatted\_full\_name, 0, 7}; // Stephen

    string last\_name = unformatted\_full\_name.substr(7, 7); // Hawking

    string formatted\_full\_name{};

    formatted\_full\_name = first\_name + last\_name; //StephenHawking

    formatted\_full\_name.insert(7, " "); // Stephen Hawking

#### Swap Operator - .swap

    // Entries can only include last names and they should be categorized alphabetically.

    string journal\_entry\_1{"Isaac Newton"};

    string journal\_entry\_2{"Leibniz"};

    journal\_entry\_1.erase(0, 6);

    if (journal\_entry\_1 > journal\_entry\_2) // Checking alphabetic order and re-ordering it using swap

        journal\_entry\_1.swap(journal\_entry\_2);

## Functions

* In this section, we’ll also learn how to give default parameters and how to set their values for functions to save us time and make it more efficient.
* How we can pass information to a function efficiently by avoiding copying, by using pass-by-reference.
* How we can change pass information as well as protect it from change.
* Inline functions are functions that the compiler optimizes to avoid the overhead involve when we call functions.
* Remember, when we use the auto keyword o allow the compiler to deduce the type of a variable? Well, we can use “auto” keyword to allow the compiler to deduce the type of the function return value.
  + This can save us from writing a lot of unnecessarily complex code.
* Then, we’ll learn recursive functions. These are functions that call themselves.
  + They help us to solve a class of problems that would be more complex to solve without recursion.
* For C++ STL library functions, check this page below: <https://en.cppreference.com/w/cpp/header>

### Using Functions of Some Popular Libraries

#### Cmath Library

* For further information & more useful functions: <https://en.cppreference.com/w/cpp/header/cmath>
* Don’t forget to #include <cmath>

   double num {};

    cout << "Enter a number (double): ";

    cin >> num;

    cout << "Sqrt of " << num << " is: " << sqrt(num) << endl;

    cout << "Cubed root of " << num << " is: " << cbrt(num) << endl;

    cout << "Sine of " << num << " is: " << sin(num) << endl;

    cout << "Cosine of " << num << " is: " << cos(num) << endl;

    cout << "Ceil of " << num << " is: " << ceil(num) << endl;

    cout << "Floor of " << num << " is: " << floor(num) << endl;

    cout << "Round of " << num << " is: " << round(num) << endl;

    double power {};

    cout << "\nEnter a power to raise " << num << " to: ";

    cin >> power;

    cout << num << " raised to the " << power << " power is: " << pow(num, power) << endl;

Outputs:

Enter a number (double): 4.4

Sqrt of 4.4 is: 2.09762

Cubed root of 4.4 is: 1.63864

Sine of 4.4 is: -0.951602

Cosine of 4.4 is: -0.307333

Ceil of 4.4 is: 5

Floor of 4.4 is: 4

Round of 4.4 is: 4

Enter a power to raise 4.4 to: 4

4.4 raised to the 4 power is: 374.81

#### Random Number Generation with <cstdlib>

* In order to use random numbers, we’re using the rand() function which is included in <cstdlib>.
  + rand() function has been used a long time in C++ and in C as well.
  + But there’s a better way to create random numbers if you’re really doing statistical processing and that’s the <random> header file.
    - But for now, we won’t go into details on that, we’ll just use <rand>
    - But if you really need very precise distributions, consider using the <random> header file.
* We’ll also include <ctime> library, which is required for time(). We’ll talk about this later.
* In order to use random numbers, we really need to seed a random number generator.
  + A random number generator is just a generator that creates random numbers.
* Computers are real bad when creating random numbers. We have pseudorandom numbers. That means is that the numbers themselves are random but their generation sequence is not random.
  + So in order to create more real random numbers, we want to seed that random number generator with a different value (based on the captured system time) each time that way it creates different sequences of values.
  + We’ll both cover the examples with seeding and without seeding to see the difference.
* RAND\_MAX could change based on the system. But it’s guaranteed to be at least 32767.
* Our random number generation seed code is like this:

    srand(time(nullptr)); // this seeds the random number generator based on the system time.

* And the random number generator code line like below

        random\_number = rand() % (max - min + 1) + min; // generate a random number [min, max]

* Works by returning a number between 0 and RAND\_MAX (32767 for us).
* To scale the result, we’re taking the max formula based modulus of rand() result then add min to it.

Here’s our full code:

    int random\_number {};

    size\_t count\_of\_randoms {10};

    int min {1}; // lower bound (inclusive)

    int max {6}; // upper bound (inclusive)

    // Seed the random number generator

    // If you don't seed the generator, you'll get the same sequence random numbers at every run

    cout << "RAND\_MAX on my system is: " << RAND\_MAX << endl;

    srand(time(nullptr)); // this seeds the random number generator based on the system time.

    for (size\_t i {0}; i<count\_of\_randoms; i++)

    {

        random\_number = rand() % (max - min + 1) + min; // generate a random number [min, max]

        cout << random\_number << " ";

    }

    cout << endl;

And here’s the example result:

RAND\_MAX on my system is: 32767

2 1 6 1 1 1 4 3 5 1

* The outputs change everytime the program run again. If we comment the srand function, it’ll give the same patterned random numbers every time.

### Declaring Functions & Function Prototypes

* There’s one caveat about function definitions.
  + The compiler must know the function details that is the function name, parameters and so forth before the function is called.
  + Let’s look at the example below:

A white background with black text

Description automatically generated

* For both keeping function declarations above the main function to not cause complications and not getting an “undefined” error, we use function prototypes. We can place these prototypes above where that function is needed and therefore we tell the compiler that function exists, what it expects and gives, where it can find it’s definition, like a demo.

A close up of text

Description automatically generated

* When using function prototypes, both two prototype examples can be used. Because supplying the name of the parameter is optional. Compiler won’t use the parameter name, it’ll only use it’s type, which is int.
  + But anyways, best practice is to provide the parameter names for documentation purposes. It helps to identify and group functions more easier especially if they’re declared and prototypes are used in another source and header files.
* The protoype’s both outputs and inputs must match the exact output and inputs at function declaration.

### Function Parameters and Outputs

A white background with black text

Description automatically generated

* In above example, function expects a C++ string object. The “Frank” word on line “say\_hello(“Frank”) in main function is a C-style string literal. But the function expects a C++ string object. These types are different.
  + In this case, even if the types are different, if compiler knows how to make the function call works, it’ll try to convert one input type to another. In this case, c-style string literal “Frank” is converted to a C++ string object.
  + The same conversion would happen if a function expected a double and we pass an integer to it.
    - Integer would be promoted to the double as we’ve seen before.

#### Pass by Value

* In C++, when you pass data into a function, that data is normally passed by value.
* Pass by value means that the value of the data is passed in by copying. So the compiler makes a copy of the data.
  + The code in the function doesn’t change the argument’s original data.
  + It just changes the copied one.
    - Because the copy is inserted on another location on the memory rather than the original value’s location.
* Pass by value is sometimes good and sometimes not so good.
  + Good because we’re making a copy, we can’t change the original argument by mistake or intentionally.
  + Bad because sometimes making a copy of the data can be expensive, both in storage, memory and time for copying and extra storing.
  + Bad when sometimes we really do want to change the actual data being passed in.
* C++ has solutions for all these use cases above and we’ll learn about Pass by reference soon.
* Let’s define some vocabulary used in programming languages, that’s formal vs. actual parameters.

##### Formal Parameters

* Formal parameters are the parameters defined In the function’s definition, in the function header.

##### Actual Parameters (Arguments)

* Actual parameters are the parameters that are used when the function is called, that are present in the function call statement.
* Many programmers and languages call actual parameters “the arguments” to a function.
* In C++, actual parameters are passed by value or copied to the formal parameters.
  + Here’s an example below:

A white background with black text

Description automatically generated

Another example:

    int num {10};

    cout << "num before calling f\_pass\_by\_value\_1: " << num << endl; // 10

    f\_pass\_by\_value\_1(num); // becomes 1000 inside function

    cout << "num after calling f\_pass\_by\_value\_1: " << num << endl; // 10 again, didn't change

void f\_pass\_by\_value\_1 (int num)

{

    num = 1000;

}

### Function Return Statement

* If a function’s return type is void, then the return statement is optional.
* A return statement can occur anywhere in the function body, but it’s usually at the last statement of the function body.
  + Because the return statement immediately terminates the function.
* It’s possible to have multiple return statements in a function, however this is not the best practice. You typically want a single return statement.

### Default Argument Values

* As we’ve seen, when we call a function, we must provoide all the arguments that function requires, and they must be in the right order and the right type.
* But sometimes when we call functions, some of the argument values that we pass in tend to be the same values most of the time.
  + For example, if we have a function that calculates the cost of an item, we can provide the function with the base cost of the item and the sales tax rate.
    - Suppose that 98% of our customers live in a region where the tax rate is 6%. That means we have to provide the tax rate in every function call, even though it’ll almost always be 6%.
* C++ allows us to provide default values for arguments.
  + So in the case of the sales tax, we omit (çıkarmak) the argument from the function call, then the C++ compiler will automatically replace it with the default value of 6%.
  + However, if we have a customer whose sales tax rate is 8%, then we can provide it explicitly and override the default value.
* As we’ll see in the next few sections, we can add default arguments to the function prototype or the function definition but NOT BOTH.
  + Best practice is to do it in the function prototype.
* Default arguments must appear at the tail end of the parameter list. We can also have multiple default arguments.

Now, let’s see an example

A screenshot of a computer code

Description automatically generated

#### Downside of Default Argument Values

* Once we omit a paramters and use a default, then all the remaining parameters must also be omitted and their defaults used.
* Remember that default argument values that do not change should be placed at the tail end of the parameter list, and those which change most often should be placed at the beginning.

For example:

I have a question in mind. Say for a function having three parameters, each with default value. When calling the function, do we have a choice to specify the later inputs only, for example the second and third parameters, without specifying the first one to ask the compiler to take the default value for the first parameter?

Some sample codes from the exercise, for example I want to call the function :

//prototype

void print\_grocery\_list(int mangos=13, int apples=3, int oranges=7);

//main

int main() {

**print\_grocery\_list(,4,8)**;}

//function

void print\_grocery\_list(int mangos, int apples, int oranges) {....

}

for example if I want to call the function with default number of mangos, 4 apples and 8 oranges. How can I do that?

Answer:

No, we can't do that.

Once we omit a paramters and use a default, then all the remaining parameters must also be omitted and their defaults used.

    greeting("Joel",,"Jr."); // We can’t do it. We must provide a modified input for 2nd parameter before reaching to 3rd.

* To get closer to doing it, we must change the order of the input parameters.
* Remember that default argument values that do not change should be placed at the tail end of the parameter list, and those which change most often should be placed at the beginning.

### Overloading Functions

* In C++, we can have functions with different parameter lists that have the same name.
  + For example, we may have many ways to display information to the screen depending on what we want to display.
  + So rather than having many functions with different names such as display char, display int, display double and so forth, we can have a single name display, and then implement “different versions of the function” for each type of parameter.
  + Then we just let the compiler figure out which function to use based on the function call arguments and the defined function parameters.
    - It may sound compilated but it’s much easier.
* This is a great use of abstaction since as a developer all we need to think is display and pass in whatever information we need.
  + We don’t have to keep track of dozens of different function names.
* In software engineering, we have a principle called “polymorphism”, which means many forms for the same concept. This is an example of polymorphism.

Now let’s see what overloaded functions look like:

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

* If we add numbers with two integers, then the “int, int” version of the function is called.
* If we call add numbers with 2 doubles, then the “double, double” version of the function is called.
* It’s important to understand that we must implement all of the overloaded versions. Notice that the code for these 2 functions is nearly identical except their input types.
* C++ has a feature called function templates that allow us to just write one generic version of the add\_numbers function and it’ll take care of providing the correct version when called.
  + Function templates are a little bit more advanced topic and we’ll discuss it when we talk about the STL library.

Now let’s look at another example:

A close-up of a code

Description automatically generated

* The parameter list for these functions must be different so the compiler can tell them apart. Once these functions are implemented, we can call display and pass in our data.
* The compiler will check the type of the argument in the function and match it to one of the available overloaded display functions.
  + If it can’t match it or if it can’t convert the argument to one that matches, then we get a compiler error.

Here’s a full example:

//.........................................Function Overloading

void print(int num);

void print(double num);

void print(string s);

void print(string s, string t);

void print(vector <string> v);

    //.........................................Function Overloading

    print(100); // int

    print('A'); // int 65 // because characters are always promoted to integers, compiler takes it as integer because we don't have a display char overload.

    print(102.3); // double

    print(102.3f); // double // F makes it float and float is promoted to double because we have no function that takes float.

    print("C-style string"); // C++ string // actually an unvalid parameter because we only defined a function for string vectors, not C-style strings. But because C++ compiler knows how to convert a C-style string to a C++ string object, so it'll use that conversion.

    string s {"C++ string"}; // C++ string

    print(s);

    print("C-style string", s); // both C++ string // C-style string is converted to a C++ string

    vector <string> three\_stooges {"Larry", "Moe", "Curly"};

    print(three\_stooges); // vector of strings

//...........................................Function Overloading

void print(int num)

{

    cout << "Printing int: " << num << endl;

}

void print(double num)

{

    cout << "Printing double: " << num << endl;

}

void print(string s)

{

    cout << "Printing string: " << s << endl;

}

void print(string s, string t)

{

    cout << "Printing string 1: " << s << ", string 2: " << t << endl;

}

void print(vector <string> v)

{

    cout << "Printing vector of strings: ";

    for (auto s: v)

    {

        cout << s + " ";

    }

    cout << endl;

}

#### Restriction of Function Overloading

##### Output Error

* There is one restriction to function overloading.
* The return type is not considered when the compiler is trying to determine which function to call.
* In example below, we have 2 overloaded functions, both called with same name and both expect no parameters. The only difference is their output types.

A white background with black text

Description automatically generated

* This’ll produce a compiler error since the only difference is the return type.
  + Consider the output statement at the bottom line. Which function would the compiler call? It could call either and that’s the problem, it won’t guess. So i’ll produce a compiler error.
* Overloading functions is used extensively in object-oriented design, and we’ll see it again when we design our own C++ object-oriented classes.

##### Overloading with Default Arguments

* Also, be careful when you overload functions and you use default arguments. If you use a default argument and you don’t give an input parameter when calling the function, the compiler will focus on the function declaration with the default argumented one and try to use it if no parameter is supplied when calling that function.
  + A more complicated similar state is, when you give 2 default arguments to 2 different overload declarations and then call the function with empty parameters, then the compiler can’t decide easily which one to use and will give you a compiler error like “call of overloaded \_function\_ is ambiguous (belirsiz)”.
  + So whenever you use overloaded functions with default arguments, you need to be precise and careful.

### Passing Arrays to Functions

* We can pass arrays into functions by providing the square brackets in the parameter information.
  + void print\_array (int numbers [] };
* So in this case we can have a function named print\_array that expects an array of integers and returns nothing.
* The idea is that this function iterates through the array and displays array values. May seem pretty handy but arrays are different in C++. Remember that an array name evaluates to the address or location and memory of the first element of the array, in other words, it points the beginning of the array.
* So what’s being passed into the function is not a copy of the entire array like we learned in “pass by value”. The passed information is only the address of the first element of the array.
  + That means the function has no idea how many elements are in the array. So the programmer doesn’t know how many times to iterate after it’s passed into the function.
* So when we pass arrays to functions, we also need to pass in the size of the array, so that we now know how many times to iterate.

Let’s show it in an example.

A screenshot of a computer code

Description automatically generated

* In above example, you can see that both the function prototype and the function definition have square brackets after the formal parameter name. This tells the compiler that the function expects an array of integers in this case.
* When we execute the function, we have no idea how many times we need to iterate.
  + Since the array name has no size information, we’re stuck.
  + If the array has a sentinel (bitiş simgesi) value, like a nulls in c-style string, then we can iterate until we see that sentinel value. But in this case, my\_numbers array has no such sentinel value.
  + There’s no way we can write this function body in a way that it will work with any array of integers.
* So the solution is to pass in the size of the array to the function as well. We’ll do that below:

A white background with black text

Description automatically generated

* Now we can easily write a for loop that iterates through the array and displays every integer and stops at where it should.

#### A Potential Issue (Changing Original Array)

This looks great but there is one potential issue that we need to be aware of:

* Since we’re passing into the function, the location of the actual array that was declared in main, that means we can modify that actual array from within the function.
  + This doesn’t seem like “pass by value” because the original array’s content changes inside the function, because we’re dealing with addresses.
  + But at some point it’s still pass by value because it’s copying the starting address of the array. It still copies and uses that address but it can change the value on that address and at the following addresses.
* This could be useful in the case of a function like “0 array” that we can call whenever we want to 0 out all the elements in an array.
  + However, in the print\_array function, we dont want to modify the array. If we do, it’s probably an unintentional error.
* We’ll see how to protect ourselves from this kind of error in the next slide, but first let’s see how we could write a useful function like 0 array.

A white background with black text

Description automatically generated

* Let’s see what we can do to have the compiler help us, so we don’t change an array we don’t want to.

##### const parameters

* We can define function parameters as const parameters. This tells the compiler that these parameters are read-only within the function body. Any attempt to modify them will result in a compiler error.

A white text with black text

Description automatically generated

* We don’t declare array as const at the initialization and declaration stage. We just make the function parameter as constant and that’ll be enough. See below example:

void print\_array(const int array[], size\_t array\_size) // prevent changes in the array by making the parameter constant

{

    cout << "Printing array elements" << endl;

    for (size\_t i {0}; i < array\_size; i++)

    {

        // array[i] = 500; // error, can't change it because it's a constant

        cout << "Element " << i << ": " << array[i] << endl;

    }

}

    int numbers[] {0, 1, 2, 3, 4};

    print\_array(numbers, 5);

### Passing Vectors into Functions

* When using vectors, we don’t have to provide an additional array length input parameter like we did in arrays. Since vectors automatically contain a size data inside them, we can directly use it in loops by doing vector\_name.size().

void print\_vector(vector <int> array)

{

    cout << "Printing vector elements" << endl;

    for (size\_t i {0}; i < array.size(); i++)

    {

        cout << "Element " << i << ": " << array[i] << endl;

    }

}

    vector <int> numbers\_v {0, 1, 2, 3, 4};

    print\_vector(numbers\_v);

### Pass by Reference

* So far we’ve seen that when we pass a parameter into a function, it’s passed by value by default. That means a copy of the actual parameter is made. We also saw that when we pass an array into a function, we don’t make a copy, instead we use the location of the array, which means we can change the original content from within the function.
* Sometimes we want to be able to change an actual parameter from within a function. As we saw with arrays, this is achieved by passing in the location of the actual parameter.
  + We can do this in C++ for any variable types, not just arrays by using reference parameters.
* Reference parameters create an alias. So now the formal parameter in the function is an alias to the actual parameter.
  + No copy is made. When you change the formal parameter, you’re changing the actual parameter.
  + This is called pass by reference and it’s really easy to achieve in C++ by using the ampersand (&) symbol in the parameter list.

Now let’s see an example:

A white background with black text

Description automatically generated

* Notice that the ampersand (&) sign is only used as a function parameter in the declaration and prototype. We don’t use & sign when calling the function or when declaring the input variable (number).
* Notice that the parameter list to the function is not an ordinary integer, it’s a reference to an integer thanks to ampersand.
  + This means that “num” is an alias for “number”. If we change “num” inside the function, we’ll change “number”, too.
* Behind the scenes, the location of numbers are being passed into the function.
* While pass by reference is super useful, we have to be careful at couple points which’ll be discussed.

#### Advantages and Use-Cases

Pass by reference is super useful for several reasons.

1. It allows us to change an actual parameter if we need to.
2. We don’t make a copy of the parameter which could be large and take time. We save time and space.

#### More Examples

void vector\_organizer\_1(vector <int> v)

{

    // for (auto num: v) // Doesn't work properly, don't use this until you learn properly

    // {

    //     v.at(num) = 2;

    // }

    for (size\_t i {}; i < v.size(); i++)

        v.at(i) = 2;

    cout << "Elements after changing the vector v1 (inside func): ";

    for (auto element: v)

        cout << element << " ";

    cout << endl;

}

void vector\_organizer\_2(vector <int> &v)

{

    for (size\_t i {}; i < v.size(); i++)

        v.at(i) = 2;

    cout << "Elements after changing the vector v2 (inside func): ";

    for (auto element: v)

        cout << element << " ";

    cout << endl;

}

    vector <int> v1 {1,2,3,4,5};

    vector <int> v2 {1,2,3,4,5};

    cout << "Elements before changing the vector v1: ";

    for (auto element: v1)

        cout << element << " ";

    cout << endl;

    cout << "Elements before changing the vector v2: ";

    for (auto element: v2)

        cout << element << " ";

    cout << endl;

    vector\_organizer\_1(v1);

    vector\_organizer\_2(v2);

    cout << "Elements after changing the vector v1 (after func): ";

    for (auto element: v1)

        cout << element << " ";

    cout << endl;

    cout << "Elements after changing the vector v2 (after func): ";

    for (auto element: v2)

        cout << element << " ";

    cout << endl;

Output:

Elements before changing the vector v1: 1 2 3 4 5

Elements before changing the vector v2: 1 2 3 4 5

Elements after changing the vector v1 (inside func): 2 2 2 2 2

Elements after changing the vector v2 (inside func): 2 2 2 2 2

Elements after changing the vector v1 (after func): 1 2 3 4 5

Elements after changing the vector v2 (after func): 2 2 2 2 2

* In function v1 code, we’re using pass by value. We’re making a complete copy of the vector object in the function. So the formal parameter “v” will be a copy of the actual parameter “data” in this case. In order to make a copy, we need to allocate storage and copy values over. If the vector is large, we could run into performance issues.
  + We just easily fixed it in function v2 code by just adding ampersands on the parameter in the function declaration and prototype.
  + We avoided the storage and copy overhead of pass by value.

Another example:

void clear\_vector(vector <string> &v)

{

    v.clear();

}

    vector <string> stooges {"Frank", "Ashley", "Joe"};

    cout << "Stooges before clearing" << endl; // Frank Ashley Joe

    for (string element: stooges)

        cout << element << " ";

    cout << endl;

    clear\_vector(stooges);

    cout << "Stooges after clearing" << endl; // empty now

    for (string element: stooges)

        cout << element << " ";

#### Potential Issues and Undesired Cases

* When using pass by reference, for example we could change or entirely delete some elements of a vector passed by. To avoid this, we can use constant references as a function parameters.
  + By doing this, a non-constant defined data will pass into the vector but it’ll be switched to a constant data when inside the vector. When exiting the vector, it becomes non-constant again.
    - Therefore we’ll also have our modularity.
  + See it on example below:

void print\_constant\_example(const vector<int> &v)

{

    cout << "Elements in the vector: ";

    for (auto element : v)

        cout << element << " ";

    // v.at(3) = 0; // Gives compiler error because vector items are constants here

    cout << endl;

}

    vector <int> v {30,30,30,30,30};

    print\_constant\_example(v); // 30 30 30 30 30 (elements can't be changed inside because of consts)

    v.at(1) = 10; // (elements can be changed here again because they're no longer consts outside the function !!)

    v.at(2) = 60;

    print\_constant\_example(v); // 30 10 60 30 30 (elements can't be changed inside because of consts)

* Using pass by reference and not letting the variable changed by accidentally by using const keyword is best of both worlds.

### Scope Rules

* When a function is called, you can think of the function as being activated. And the function parameters are bound to storage. They become alive and their lifetime is while the fufnction is executing.
  + Once the function completes, the function is deactivated and these variables and parameters are no longer alive.
  + That doesn’t mean that they’re somehow marked as unavailable. It simply means that the compiler doesn’t recognize them any longer, and the storage they were bound to will likely be reused.
* That means that the values of local variables are not preserved between function calls.
* If we have nested blocks, statements in the inner blocks can see identifiers declared in the outer blocks, but outer blocks can’t see variables defined in the inner blocks.
* However, there is one type of variable whose value is preserved between function calls.
  + This is a static local variable.

#### Static Local Variables

* Static local variable’s value is preserved between function calls.
* It’s a variable whose lifetime is the lifetime of the program BUT it’s only visible to the statements in the function body.
* These variables can come in very handy when you need to know a previous value in a function without having to pass it in all the time.
* Static local variables are only initialized once. If no initializer is provided, they’re automatically set to 0.
* A static local variable behaves like a global variable because it’s retaining it’s value, but it’s scope is local.
  + Therefore, compared to global variables:
    - It’s more immune to errors, you’ll know exactly where to look in case of errors.
    - It’s more memory space and time-saving
    - It’s more efficient.
* That’s why we should avoid using global variables and use static local variables whenever we can if we want to retain a variable’s value.

#### Global Scope and Global Identifiers, Global Variables

* Identifier declared outside ANY function or class.
  + main is also a function and global variables are declared even before main.
* Global identifiers are visible to ALL parts of the program after the line it’s declared.
* Best practice is:
  + You can use global constants.
  + AVOID USING global variables.
* If the variable is a global variable and needed to be used in a function, we don't have to pass it to the function. The code in the inner functions has direct access to the global variable. So it’s not copied by using pass by value.
  + Global variables’ values can be changed without the need of passing them into functions, as long as they’ve been declared before the location where it’s used.

#### Examples

Let’s look at some key examples.

##### Creating and Reviewing Different Scopes

First one is about local scope rules which is interesting and informative about creating a new scope easily with empty curly brackets:

    int num {100};

    int num1 {500};

    cout << "Local num is: " << num << " in main" << endl; // 100

    {   // Create a new level of scope

        int num {200};

        cout << "Local num is: " << num << " in inner block of main" << endl; // 200

        cout << "Inner block in main can see out - num1 is: " << num1 << endl; // 500

        // num1 became 500 even at the inner because it couldn't find a closer declaration for num1.

// Then it tried to find it at the outer scope and found it.

        int num1 {700};

    }   // End of new scope

    cout << "Num1 is: " << num1 << " in main after scope1" << endl; // 500 again

    {

        int num1 {900};

    }

    cout << "Num1 is: " << num1 << " in main after scope2" << endl; // 500 again

* When finding a variable, compiler tries to find it first at closest local scope. If it couldn’t, it scans outer scopes until it finds it.

##### Global Variable Basics Example and Changing It

void global\_example()

{

    cout << "\nGlobal num is: " << num << " in global\_example - start" << endl;

    // 1st call: 300, 2nd: 600, 3rd: 1200, 4th: 2400

    num = num \* 2;

    cout << "Global num is: " << num << " in global\_example - end" << endl;

    // 1st call 600, 2nd: 1200, 3rd: 2400 4th: 4800

    // It changes the global variable's value which is declared at the most outer scope,

    // like a pass by reference example.

}

    int num {100};

    for (size\_t i {}; i<5; i++) // call the function 4 times

    {

        global\_example();

    }

Output:

Global num is: 300 in global\_example - start

Global num is: 600 in global\_example - end

Global num is: 600 in global\_example - start

Global num is: 1200 in global\_example - end

Global num is: 1200 in global\_example - start

Global num is: 2400 in global\_example - end

Global num is: 2400 in global\_example - start

Global num is: 4800 in global\_example - end

Global num is: 4800 in global\_example - start

Global num is: 9600 in global\_example - end

##### Static Local Variable Basics Example

void static\_local\_example()

{

    static int num {5000};

    cout << "\nLocal static num is: " << num << " in static\_local\_example - start" << endl;

    // 1st call: 5000, 2nd: 6000, 3rd: 7000, 4th: 8000, 5th: 9000

    num += 1000;

    cout << "Local static num is: " << num << " in static\_local\_example - end" << endl;

    // 1st call: 6000, 2nd: 7000, 3rd: 8000, 4th: 9000, 5th: 10000

}

    int num {500}; // This one will be unused because there's a closer declaration inside the example function

for (size\_t i {}; i<5; i++) // call the function 4 times

    {

        static\_local\_example();

    }

Output:

Local static num is: 5000 in static\_local\_example - start

Local static num is: 6000 in static\_local\_example - end

Local static num is: 6000 in static\_local\_example - start

Local static num is: 7000 in static\_local\_example - end

Local static num is: 7000 in static\_local\_example - start

Local static num is: 8000 in static\_local\_example - end

Local static num is: 8000 in static\_local\_example - start

Local static num is: 9000 in static\_local\_example - end

Local static num is: 9000 in static\_local\_example - start

Local static num is: 10000 in static\_local\_example – end

* Since we’ve used a static local variable, we got a variable behaves like a global variable because it’s retaining it’s value, but it’s scope is local.
  + Therefore, compared to global variables:
    - It’s more immune to errors, you’ll know exactly where to look in case of errors.
    - It’s more memory space and time-saving
    - It’s more efficient.

### How do Function Calls work in background?

* Functions use an area in memory called the “function call stack” or program stack.
  + A stack is analogous to a stack of books or a stack of dishes. If you place a book on top of a stack, then you must remove that book before removing any others.
  + This is referred to as LIFO (last in first out).
* Stacks also use the terms
  + “push” when you put an item on top of the stack,
  + “pop” when you remove an item from the top of the stack.
* In the case of a C++ program, these items are called “stack frames” or “activation records”.
* All it is, is a collection of information that represents an active function.
  + So this is where parameters are stored, local variables, the return address and more.
* Each time a function is called, an activation record is created and it’s pushed onto the call stack.
  + When the function terminates, we pop it’s activation record off the call stack, and now the top of the stack is the function that just called the one we just popped off.
* The call stack works in a very orderly manner. You can’t jumpp into or out of the middle of the stack. You must follow the LIFO rules.
  + Also remember that the call stack is finite in size. If you activate too many functions on the call stack, then it’s possible to run out of stack space.
  + This results in a stack overflow error, which is usually an unrecoverable error and your program will terminate.
* The best way to understand how function calls work is to see them visually. Let’s do some examples.
* In order to understand how function calls work, we really need to understand how memory is laid out.

A diagram of a stack

Description automatically generated

* Memory is divided into segments like
  + Code Area
  + Static & Global Variables
  + Stack
  + Heap / Free Store
* Stack is the function call stack. And this is what we’re really concered about in this section.
* We got another memory called heap or the free store. We’ll talk a lot about this area when we talk about pointers and dynamic memory allocation.
* So in this section, keep in mind that what we’re talking about is the function call stack area. When we call functions and they finish and they pop off the stack, all happens in this stack area of the memory.

#### What happens when main function calls another function?

* Although there are other ways to achieve the same results, here’s the typical one.
* Suppose we have a main and main calls a function called func1.
* Everything here, push and pop operations are happen in the stack.

Main side:

1. Push empty space (on the stack) for the function’s return value
2. Push space for the parameters
3. Push the return address
   1. That’s very important because func1 needs to know where to come back to when function is finished executing.
4. Transfer control to func1 (jump instruct on assembly)

Func1 side:

1. Push the address of the previous activation record.
   1. That’s basically moving a stack pointer. That way you know where the top of the stack is.
2. Push any register values that need to be restored before returning to the caller
3. Perform the code in func1
4. When the code is finished, restore the register values that way main is where it was before.
5. Restore the previous activation record
   1. Move the stack pointer by popping all this stuff off the stack
6. Store any function result where main wants it (main already gave this address)
7. Transfer the control to the main by jumping back to that return address that main pushed before.

Main side after func1 is done:

* Now, main knows where the parameters are and it knows where the return values are.

1. Pop that information off the stack to clear up the stack.
2. Pop the return value (grab return values if you can).

* So that are the basic steps. Now we’ll walk through a code example by additionally drawing a function call stack by doing it. So we can see exactly what’s going on from the parameter perspective.
  + For now, we won’t worry about stack pointers, static links, dynamic links and all the stuff that’s on an activation record.
  + But more generally, let’s worry about the parameter passing so we can really understand what’s going on because it’s important that we understand this, especialy when we learn about recursion later on.

Function declarations:

int func1 (int a, int b)

{

    int result {};

    result = a + b;

    func2(result, a, b);

    return result;

}

void func2 (int &x, int y, int z)

{

    x += y + z;

}

In main function:

    int x {10};

    int y {20};

    int z {};

    z = func1(x,y);

    cout << z << endl;

* Program execution starts at main. So, main is going to be activated. It’s got a function activation record because it’s a function. So let’s assume main is already on the stack.
  + Main has an x, y, z variables and their initialized values so we need space for those local variables.
* When we call func1, main stops what it’s doing. It allocates space for x,y. Then it pushes an activation record on here for func1.
* Func1 is now in another segment of stack, not in main. And we need space for a, b and result. We’re doing pass by value here. Therefore x copied to a, y copied to b.
* Then it starts executing func1. Does the sum calculation and assigns it to result.
* Then before it calls func2 and pass result, a, b to func2, it stops what it’s doing there. It activates func2 at a different segment of stack (by the time all along, those stack pointers are moving upper and upper).
* Now it’s in func2, it needs space for x, y and z. But notice x is a call by reference parameter. So we’ll have to deal with that. So after making copy operations for y and z, y becomes 10 and z becomes 20 in func2.
  + Now let’s talk about that reference parameter x. Remember that reference parameters are aliases. So x is an alias for result. So any changes and updates made to x will also be automatically applied to result, too. That’s the whole point of pass by reference.
* Now it starts executing func2. x(result) = x (result) + (y + z). (y+z) becomes 10 + 20 = 30. And then we add x (result, which was 30) to it. x is a reference to result. Result was calculated 30 from func1. So 30 + 30 = 60 is the new value of x, which’ll also change the result because it’s a reference parameter. So both x and result becomes 60.

Here’s a snippet of basic stack before it’s popped off:

A white board with writing on it

Description automatically generated

* Now, func2 is totally finished. So it gets popped off the stack with their variables.
* It gets back to func1. We’re back from the function call. So it needs to return result. Then result (60) will be assigned to z.
* Func1 is also totally finished, so it pops func1 and it’s variables from stack. We only got main function in the stack (basically).
* From here, main has only 1 statement left to execute which is displaying z. After displaying it and returning 0 if everything is successful until here, main function is also done executing. Therefore main function and it’s variables also pops off from the stack.
* We skipped a lot of these on purpose because there’s a lot to draw and it gets really, really cluttered. But think about all this as function call overhead. We really don’t have to worry about pushing registers and transferring control. All that’s done for you by the compiler. But there’s a certain amount of function call overhead here.
* We’ll review this again when we talk about recursion. Recursion is like a function calls a function calls a function calls itself etc. So it’s really important to understand what’s going on with the stack call frame. So we really know what’s what’s being passed and what’s being returned.

### Inline Functions

* As we saw in the previous video, function calls have a certain amount of overhead. We need to create an activation record, push it on the stack, deal with parameters, pop off the activation record when the function terminates and deal with the return addresses and return values.
  + Although all of this can happen very quickly and efficiently, it still happens.
* Sometimes we have a very simple function and the function call overhead might be greater than the time spent executing the function.
  + In cases such as these, we can suggest to the compiler that it generate inline code.
* Inline code is basically inline assembly code that avoids the function call overhead.
* Inline code is generally faster.
  + But if you inline a function many times, then you’re duplicating function code in many places and it could lead to larger binaries.
* That said, compilers are so sophisticated now that they’ll likely inline code even without your suggestion.

Let’s see what an inline functions looks like:

inline int add\_numbers(int a, int b)

{

    return a + b;

}

in main:

    int result {};

    result = add\_numbers(100, 200);

* We simply precede (önüne koymak) the function return type with keyword “inline”.
* Inline functions are usually declared in header or .h files since the definition must be available to every source file that uses it.
* As mentioned earlier, compilers are so good now that most will make sure functions like this become inline anyway even if you don’t provide the “inline” keyword.
* Don’t worry too much about asking the compiler to inline your functions, but now you know what inline functions are in case you see them in C++ code out there.

### Recursive Functions