# C++ Pro: V5 Essentials

## Guide to Understanding the C++ Language for VEX EDR

Evan Hess

# Preface

Programming is an art. Programming is a skill. Programming is about the magic of bringing something to life out of nothing. When I program, I strive to see the joy on my teammates' faces when we finally bring our robot to life and win that match. Without you, the programmer, that will never happen. Your job is to help your ideas and other's ideas come to fruition. It may be a thankless task, and when something electronic goes wrong, guess who they're going to blame. But through all of that, your code will be one of the most important factors to success.

This guide was constructed to help VEX EDR students of any grade-level who want to understand the V5 C++ Pro option to have the utmost control over their code. This is NOT an advanced guide, nor will we go over every small nuance in the C++ language. This is specifically for immediate use with VEX EDR C++ code in order to quickly get a functional robot up and running and teach some computer science concepts in the process. Though programming can be done in multiple other languages using third party applications and/or websites, for example Robot Mesh Studios, C++ will almost certainly be the standard, and thus the topic of this guide. For complete understanding, it's recommended that the reader have experience in RobotC (or C in general) or similar high-level languages (e.g. Java).

Complete projects will be hosted on GitHub to download at the following link: https://github.com/vexGuide/vex-v5-guide.

This guide will use the following notations to identify key elements in this text:

***Note:*** *Italicized text* will be used to explain extra topics or ideas that are usually not essential yet will most likely not harm the ability to understand both the guide and the code or text it refers to.

`//Code in this font` is used when code or code snippets are introduced; therefore, some of the most important elements of the text are used in this style.

In this guide, we will specifically use VEXcode V5 Text program that, at the time of writing, can be found here: https://www.vexrobotics.com/vexcode-download#V5Text.

This guide should not be the end of your endeavors. To start, go check out the VEX EDR Forums. They have people working to solve your problems and sometimes they post their own stuff too! Look at the API, which I provide a link to in the GitHub repository readme file. You might find some neat stuff in there, like rumble commands! But afterward, search up tutorials or examples on C++, and learn some more tips and tricks. VEX was never meant to be the end, it was always supposed to be the beginning.

Finally, I do sincerely hope you learn something new, or gain a better understanding of VEX's C++ Pro option. Like I previously said, C++ is how the future of VEX Robotics coding will be. Choosing the Pro option should the ultimate end goal of all VEX programmers. Visual supplements almost always have some drawback—whether that be how it essentially hides code, or how it will never be exactly how you want it. This guide is, once again, NOT a full tutorial on C++ in general, but it can be the first step towards mastering the C++ language; and my ultimate goal is to foster an environment that will help you in enjoying the magic of programming.

---

# 1. Skeleton

In the VEXcode editor, make a new empty C++ project (do NOT check the competition box) titled **MyComp**. This first section will seem strange at first, but this will be dissected bit by bit. Remove all code from the **main.cpp** and type the following:

```
#include “vex.h”
using namespace vex;

brain Brain = brain();
competition Competition = competition();

void pre_auton(void) {

}

void autonomous(void) {

}

void usercontrol(void) {
   while(true) {
      task::sleep(20);
   }
}
```

```
int main(void) {
   Competition.autonomous(autonomous);
   Competition.drivercontrol(usercontrol);

   pre_auton();

   while(true) {
      task::sleep(100);
   }
}
```

Long, I know; while most of it is empty, there is plenty to comprehend. Let's start with the first two lines. **`#include "vex.h"`** essentially copy-pastes code into the program from the header file "vex.h" when the code is compiled. A header file contains the names for functions that you can use in your code that are either defined by you or someone else. We won't need to worry about what the header files contain; they just make all this code supplied by VEX possible. **`using namespace vex;`** is written to make writing code easier, specifically the next lines: **`competition Competition;`** and **`brain Brain;`**. These two lines would need to be prepended with **`vex::`** in order to declare these two statements without **`using namespace vex;`**. This is because **`namespace`** is a keyword in C++ which tells the compiler (the application that converts C++ into machine-readable form), that **`vex`** code defined by someone else is now part of this program.

The proceeding lines declare **objects**. Objects in C++ are collections of data that represent real-life objects (shocking, I know). "competition" and 'brain" are defined by VEX in a header file. These two, unsurprisingly, control the competition and brain functionality,

respectively. Later, the brain will become more important, and, for our purposes, the competition is nearly useless besides the vital two lines of code near the end.

The following three code blocks are called **functions**. They are blocks of code that are given a name and execute a series of statements. You can reuse these functions as many times as you want. Usually, they either are reusable statements, or make code more readable by grouping similar statements.

The first two functions do nothing, because we haven't put anything yet. If you've used RobotC before, these functions should look familiar. The first one, **`pre_auton(void)`**, is used on robot startup (we will see where it's called at the end). The second, **`autonomous(void)`**, is the automatic part of the robot's functionality. We'll use some basic commands to get you started and some examples of sensor use. The one we will focus on right now is the **`usercontrol(void)`** function. This describes the main focus of our functionality and is the easiest to implement after the underlying code is explained.

In this function, there is a loop called a **while** loop. Loops have some condition that must equal **true** (in other words, it must be valid) for the statements within its braces ("{ }") to execute. Until this condition is **false**, the statements will continue to execute. In our case, it says true, so therefore it will never be false. Because it is never false, it will execute FOREVER.

In this loop, there is nothing special and we'll get rid of it soon. **`task::sleep(20)`** means that the current task (which is the program because there are no other tasks running) will pause, or sleep, for 20 milliseconds. This is just so you don't waste resources, but you

almost never need it for the **`usercontrol`** function since it works until the end of the match or you stop the program.

In every executable C++ program, there must be a main function that the compiler can run so the program may begin. In this main function, the **`competition`** object returns! These next two lines tell the object that when the judges commence the match, the **`autonomous`** and **`usercontrol`** functions are used as variables to set their respective function. When the match starts, these two methods are called, in order, and your function begins.

The final lines call the pre-autonomous function and prevent resources from being wasted, like last time. The only difference is that this time, the tasks sleeps for 100 milliseconds, or one-tenth of a second.

The end has come! Finally! This will always be your competition template. You can click the check box to auto generate this, but this is the cleaned-up version, showing you the bare essentials. The most important part is to NEVER MODIFY THE MAIN FUNCTION, unless you know EXACTLY what you're doing. Editing the main function could lead to inoperable robot code.

# 2. Constructs

C++ has language constructs that help define what you can do in it. This ranges from variables to namespaces to classes. Let's start with the basics: variables.

## 2.1 Variables & Data Types

C++ has several types of ways to store data. These are called **data types**. All these types are made of bits. Each bit has two states: 1 or 0. All data types have some limit. This table explains the typical sizes visually (these ranges vary from machine to machine):

| Data Type | Number of Bits | Representation | Range |
|---|---|---|---|
| bool | Unspecified | Unspecified | 1 or 0/true or false |
| char | 8 (1 byte) | $2^8$ | -128 to 127 |
| short | 16 (2 bytes) | $2^{16}$ | -32768 to 32767 |
| int | 32 (4 bytes) | $2^{32}$ | -2,147,483,648 to 2,147,483,647 |
| long | 32 (4 bytes) | $2^{32}$ | -2,147,483,648 to 2,147,483,647 |
| long long | 64 (8 bytes) | $2^{64}$ | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 32 (4 bytes) | $2^{32}$ | About $1.5x10^{-45}$ to $3.4x10^{38}$ |
| double | 64 (8 bytes) | $2^{64}$ | About $5.0x10^{-345}$ to $1.7x10^{308}$ |
| long double | 64 (8 bytes) | $2^{64}$ | About $5.0x10^{-345}$ to $1.7x10^{308}$ |

The first one, **bool** is a Boolean value, which just means it can hold either **true** or **false**, or 1 or 0, meaning true or false, respectively. It's mostly used in conditional statements. The next, **char** holds numbers, yet it stores them in a format (ASCII, Unicode, UTF-8, etc.) and prints out the respective value. The following four types are either int or variants of int; they hold standard integers and have a range that varies. C++ allows you to skip writing **int** at the end of each prefix (short or long), so you can declare variables as "**short a**" instead of "**short int a**". The final three are similar, but they are **floating-point** integers. This essentially means that they can hold decimal numbers. It is a BIG range, but they can be exceeded by too many decimal places below $10^{-1}$.

The proceeding lines explain how you can declare—when you create an object—and initialize it—when you give it a value for the first time:

```
int a; //declare the variable
a = 10; //initialize the variable
char b = 5; //declare and initialize
long long c = 8273527;
bool isRunning = true;
float decimal = 10.5f;
//the "f" can be used to explicitly say that it's a float
```

***Note:*** *"//" denotes a comment, you can use this in your own code! Anything after that point on that line is ignored. Essentially, it's a note for you to look back on. ";" follows almost every line of code; if in doubt, use the semi-colon. The compiler will let you know if it was not expected, in which case you must remove it.*

One of the formats for declaring or initializing variables in C++ is: **type name** or **type name = value**, respectively. This is one of the most important parts of C++. You can change the value of these variables at any time by using the format **name = value**.

***Note:*** *YOU CANNOT USE A VARIABLE BEFORE YOU INITIALIZE IT, but you do not need to do so immediately, and in some cases, it's better not to.*

The last thing to mention is variable arithmetic. You can perform the basic operations of addition (+), subtraction (-), multiplication (*), and division (/). However, there is a fifth operator: **modulo**. Modulo (%) is essentially a combination of division and subtraction. It returns the remainder of a division statement. Take 10/5. That equals 2. There is no remainder. If you used % instead, it would be written as such: 10%5. The result is 0 because there is no remainder. Now let's take 22/4. The answer is 5.5, or 5 R2. Modulo takes the remainder and gives that as the answer! So 22%4 would be 2, NOT 5! Here is an example of the arithmetic.

```
#include <iostream>
int main(void) {
   int add = 10+5;     //add 10 and 5; total of 15
   std::cout << add << endl;
   //ignore "std::cout" lines, they print the result
   int sub = 10-5;     //subtract 10 by 5; difference of 5
   std::cout << sub << endl;
   int mult = 10*5;    //multiply 10 by 5; product of 50
   std::cout << mult << endl;
   int div = 10/5;     //divide 10 by five; quotient of 2
   std::cout << div << endl;
   int modzero = 10%5; //10 modulo 5; remainder of 0
   std::cout << modzero << endl;
   int modtwo = 12%5;  //12 modulo 5; remainder of 2
   std::cout << modtwo << endl;
}
```

If you want to increment or decrement a number by 1, just add **++** or **--**, respectively, to the end of the variable name. Another popular shortcut to changing the value of a number is

like this: If you want to add 5 to the current value of **`int a`**, then instead of **`a = a + 5`**, you can use **`a += 5`**.

# 2.2 Control Flow

Sometimes, you don't want code you wrote to run. Now, that may sound idiotic (after all, why write code that you won't use), but here's why you would want to do such a crazy thing.

## 2.2.1 If Statements

Let's say you want to check someone's age to see if they are eligible to drive. So, you would say, "***IF*** this person is 16 or older, ***THEN*** they can drive." In this example the person is 21, so they can drive. Here is the equivalent in C++:

```
#include <iostream>
int main(void) {
   int age = 21; //declare "age" and set it to 21
   bool canDrive; //make a bool to make sure they can drive
//notice how it is not initialized because we don't need to yet
   if(age >= 16) {
      canDrive = true;
      std::cout << "You can drive!";
      //print the words in quotes to the screen
   }
   return 0;
}
```

Here is what it means: We declare and initialize "age" and set it to 21. Then we only declare "canDrive", we do not want to initialize it, because we have no idea whether the person can drive or not; that is up to the program to decide. Now, the code has reached the **if statement**. An if statement is formatted like so: **`if(condition) { execute }`**. Notice how there is no semi-colon at the end. An if statement is executed one time and ONLY if the condition is true. If the condition equates to false, then the code within the braces is not executed.

In our example, all it does is that it checks if you are greater than or equal to (>=) 16, sets "canDrive" to true if you can, and prints out "You can drive!" if you can drive. It is simplistic, so let's add something else. The following code snippet will focus on the **if-else statement**.

```
if(age >= 16) { //previous code
   canDrive = true;
   std::cout << "You can drive!";
}
else { //note how there is no condition
   canDrive = false;
   std::cout << "You cannot drive.";
}
```

We added a new section, **`else`**. It does not check for a condition; instead, it waits to see if the condition is false. It will only execute if the condition is false and no other circumstance. If the condition is true, **`else`** will never execute its statements, and the code will continue. There is one more section to add, but we'll make a new scenario: Suppose you need to check if someone is tall enough to ride a rollercoaster, but if you're too small to ride, you can ride with a parent if you are above a "no-ride height."

```
#include <iostream>
int main(void) {
   int height = 50; //50 inches
   bool canRide;

   if(height >= 55) { //you must be at least 55 inches to ride
      canRide = true;
      std::cout << “You can ride!”;
   }
   else if(height >= 40) { //you must be at least 40 inches
      canRide = true;
      std::cout << “You can only ride with an adult.”
   }
   else { //less than 40 inches
      canRide = false;
      std::cout << “You cannot ride.”;
   }
}
```

“Else if” is essentially another “if”, with the exception that if any “if” or “else if” conditions are true in the same chain, the others will not be executed. If the height was 60, then the “else if” statements will not be executed, and the same goes for “else”. You can put as many “else if” conditionals as you want, but there must be one, and only one, “if” and, optionally, a maximum of one “else” per chain.

This table shows all the following comparison operators you can use to check the relation of values:

| Symbol | Name | Meaning |
|---|---|---|
| == | Equal to | Checks whether two values are the same |
| === | Equal to and of the same type | Checks whether two values are the same type (e.g. int) and are equal |
| != | Not equal to | Checks whether two values are not equal |
| > | Greater than | Checks whether the left value is larger than the right value |
| < | Less than | Checks whether the right value is larger than the left value |
| >= | Greater than or equal to | Same as greater than but can return true if they are equal |
| <= | Less than or equal to | Same as less than but can return true if they are equal |
| ! | NOT | Checks is the expression is NOT true |

The VERY LAST thing for this section, I promise: Sometimes you need to check if two things are happening, in which case COMPOUND OPERATORS TO THE RESCUE! Compound operators check two expressions at the same time and return true or false based on those results. Here is another table! Yay.

| Symbol | Name | Meaning |
|---|---|---|
| && | AND | Checks if both sides equate to true |
| \|\| | OR | Checks if one OR both sides equate to true |
| ^ | XOR (Exclusive OR) | Same as OR but returns true ONLY when ONE side is true |

These operators have two conditions to evaluate. Here is how to use the AND operator: **`(a == 5) && (b == 10)`**. If a equals 5 AND b equals 10, then do this….

The OR operator is used like so: **`(a == 5) || (b == 10)`**. If a equals 5 OR b equals 10, then do this…. (With an OR operator, both statements can be true and the whole expression will evaluate to true, the last operator is more like our traditional way of thinking about 'or.')

Finally, here is how XOR is evaluated: **`(a == 5) ^ (b == 10)`**. If a equals 5 OR equals 10 (BUT NOT BOTH), then do this…. The main point is that if both expressions are true, the whole expression is false because it can only be one of the two. In Computer Science and Circuitry, this is identified as "exclusive OR," hence the X standing for exclusive.

# 2.2.2 While Loops

Maybe you don't want things to run one time only. Maybe you're making a counter! Wow! How did I know you were thinking that because you totally were! Well here is the solution:

```
#include <iostream>
int main(void) {
   int a = 1; //set a to 1
   while(a < 10) { //if a is less than 10, keep going
      std::cout << a << endl; //print the value
      a++; //increment by 1
   }
   std::cout << a;
   return 0;
}
```

A while loop is basically an if statement that repeats itself until the condition equals false. It's the condition is never true, then it will never run. You can even make it infinite by setting the condition to true like so: **`while(true)`**. Look familiar? That's because we've already seen it in the skeleton program, not once, but twice! There is a variation that makes this counting job easier called a **`for`** loop.

# 2.2.3 For Loops

Let's edit that last code and change the while loop to a for loop. For loops have a different syntax to keep track of three steps: Declaration; Condition; Increment.

```
#include <iostream>
int main(void) {
   for(int i = 0; i < 10; i++) { //declare, check, increment
      std::cout << i << endl;
   }
   return 0;
}
```

Now, we have combined three steps together. First, we declare the variable ("i" is the convention in for loops), then we make the condition, and finally, we set the increment, preferably in a way that the loop will terminate. If we replaced it with **`i--`** then the loop would run forever.

These are the main loops you will need, and there are variations, which I encourage you to explore at the end of his guide.

# 2.3 Functions

Functions are, as mentioned before, blocks of code that are given a name. These functions are self-contained and if outside variables are required, they are supplied through **parameters**. These parameters are copies of the variables that are supplied by the code calling it. When the function is called, the series of statements are executed followed by a **`return`** line. Here's an example of a function without parameters:

```
#include <iostream>
void sayHello(void) { //define function
   std::cout << "Hello!"; //prints "Hello!" to the console
   return;
//this tells the compiler that there is nothing else in the function
}

int main(void) {
   sayHello(); //call (execute) the function
   return 0;
//main should always return 0 so the program exits successfully
}
```

***Note:*** *As I've said before, EVERY C++ program that is executable, there must be an* ***`int main(void)`*** *function because that tells the compiler where it must start executing your code. Other functions are put above main because the compiler must define the functions BEFORE it can call them.*

The first word in a function definition is the return type, in this case **void**, and **sayHello(void)** is the name. The "void" in the parenthesis just lets the compiler know that we're not expecting any parameters to be given when we call it, and the type "void" means that we're not giving anything back, so we will not receive anything in the calling code. Why is returning something useful? Your function is most likely setting a variable (**int a = setValue(5)**, we will see an example soon). **std::cout** is not very important to this guide, but its purpose is to print to the console, which does not work with the V5; it simply is there to make the concept easier to understand. That line, however, prints hello, and the function does not do anything else. The **return** statement signals the end of the function to the compiler. In the **main** function, we call **sayHello()**. This statement calls the function, which has all the statements in the function execute. Then finally, **return 0**. Since **main** returns an **int**, **0** is returned to whatever is running the program because it is an integer (int = integer), and it ends. Here's a very similar example using a parameter and a return type:

```
int setValue(int var) { //define a function with return type "int"
   return var; //returns an integer (var)
//end function
}

int main(void) {
     int a = setValue(5);
//calls setValue and the function returns 5, setting a to 5
   return 0; //end program
}
```

This function has the purpose of returning a value that is received. Its purpose is to show you how you can add parameters to a function and to demonstrate how functions can return some value or even objects. For now, we will stick with returning data types. This function takes **arguments** which is just some value or object that the function makes a copy of and works on. In this sample, variable **`a`** is created and calls on **`setValue(5)`** to give it a value. The function takes the 5 and returns it to **`a`** so the variable is set to 5. Easy enough, and a little too simple, but it gets the point across. The return type is **`int`**, so we must return an integer. The integer that we gave it was 5, so the function returns 5 as the integer. Make sure you return the correct return type, or else the compiler will be very, VERY mad at you. For example, if you say the return type is an integer, you cannot return a float. That goes for any return type, objects included.

# 3. User-Control

At this point, you have enough knowledge of the basics to start programming your robot! Let's go to the top, just below the **competition** line. Here we will make three global motor objects and a controller object for input. Global variables/objects are the same as variables with the exception that you can access them anywhere in the code. If you declare a variable in a function, only the function may use that variable. Here, anything can use it. Write below the **competition** line:

```
controller Driver = controller(); //if you want a partner
controller, put "controllerType::partner" in the parenthesis without
quotes
motor left = motor(PORT1);
motor right = motor(PORT6);
motor arm = motor(PORT11);
```

***Note:*** *It is a good idea to space motors out on the circuitry, specifically how, like the previous Microcontroller, ports 1-5 on the V5 are separated from 6-10, and the same applies on the other side. Don't overload any one section of circuitry, evenly distribute as much as possible across all four sections.*

We will use three motors to demonstrate basic driving with the joysticks and basic button movement with the arm. Now jump to the user-control section and clear everything in it. Here, I will show you two approaches to motor movement. Write the following:

```
void usercontrol(void) {
   arm.setVelocity(75, velocityUnits::pct); //you can set the speed
(first argument) to whatever you want, it's a percentage
   //set the speed of the left motor to the left y axis position
   while(true) {
      left.setVelocity(Driver.Axis3.position(percentUnits::pct),
velocityUnits::pct);
      left.spin(directionType::fwd);
      //set the direction forward when the joystick is pushed
      right.spin(directionType::fwd,
Driver.Axis4.position(percentUnits::pct), velocityUnits::pct);
      //or do it all in one step, this time on the right y axis
   }
}
```

If you run this code and put the motors in the right ports, you'll find that you can move the robot with your joysticks! It seriously takes one line of code to turn an immobile heap into an unstoppable VEX robot! Now, there is one more thing you can do with motors, and that is attaching them to buttons. Here's an example. Below the **`right.spin`** code, type the following, but do not go outside the while loop braces!

```
if(Driver.ButtonUp.pressing()) { //this function returns a bool
which, as we know, can be true or false
   arm.spin(directionType::fwd); //we already defined the speed
before the while loop
}
else if(Driver.ButtonDown.pressing()) //the opposite of the last if
block
   arm.spin(directionType::rev); //spin the REVerse way
}
else { //if either of those two buttons are not pressed
   arm.stop(); //stop. easy enough
}
```

This last section is self-explanatory. If the up button is being pressed, go forward (if it was real, hopefully you built it the correct way and it will go up, but if not, then just change fwd to rev), if the down button is being pressed, go backward (a.k.a., down). If neither are being pressed, stop the motor. The complete **`usercontrol`** code should look like this:

```
void usercontrol(void) {
   arm.setVelocity(75, velocityUnits::pct); //you can set the speed (first argument) to whatever you want, it’s a percentage
   //set the speed of the left motor to the left y axis position
   while(true) {
      left.setVelocity(Driver.Axis3.position(percentUnits::pct), velocityUnits::pct);
      left.spin(directionType::fwd);
      //set the direction forward when the joystick is pushed
      right.spin(directionType::fwd, Driver.Axis2.position(percentUnits::pct), velocityUnits::pct);
      //or do it all in one step, this time on the right y axis


      if(Driver.ButtonUp.pressing()) { //this function returns a bool which, as we know, can be true or false
         arm.spin(directionType::fwd); //we already defined the speed before the while loop
      }
      else if(Driver.ButtonDown.pressing()) { //the opposite of the last if block
         arm.spin(directionType::rev); //spin the REVerse way
      }
      else { //if either of those two buttons are not pressed
         arm.stop(); //stop. easy enough
      }
   }
}
```

At last, your robot is functional! Three cheers! But this is the end. Of this section! There will be a second part to his guide explaining how to use the three wire sensors, vision sensor, and autonomous code! In the process of reading this guide, you have learned basic constructs of the C++ language, including data types, conditionals, and functions. If you have any questions, comments, concerns, or suggestions, please email me at enh.robotics@gmail.com. I'm willing to listen and learn a great deal more by all, and especially those much more experienced than I. Good luck on your robotics season; and best of wishes to all of you and your programming adventures!