

# Pointers and addresses

There are two ways of declaring an object variable: either as an instance of the object or a pointer to an instance of the object. In the former case the variable holds the object and the object is created (“instantiated”) at the same time. In the latter case the variable only has space to hold the address of the object. It takes another step to create the object instance using the **new** operator and assign its address to the variable.

## Creating object instances

Location	Create object	Use the object	When the object is deleted
Variable declared inside an object, function, or block	<code>Victor leftMotor(3);</code>	<code>leftMotor.Set(1.0);</code>	Object is automatically deleted when the enclosing block exits or the enclosing object is deleted
Global declared outside of any enclosing objects or functions; or a static variable	<code>Victor leftMotor(3);</code>	<code>leftMotor.Set(1.0);</code>	Object is not deleted until the program exits
Pointer to object	<code>Victor *leftMotor = new Victor(3);</code>	<code>leftMotor-&gt;Set(1.0);</code>	Object must be explicitly deleted using the C++ <b>delete</b> operator.

There are several ways of creating object instances in C++. These ways differ in how the object should be referenced and deleted. This table shows the rules.

## Pointers vs References

C++

```
Joystick stick1(1);           // this is an instance of a Joystick object stick1
stick1.GetX();                // access the instance using the dot (.) operator
bot->ArcadeDrive(stick1);      //you can pass the instance to a method by reference
//      ...   ArcadeDrive(Joystick& j);   ...

Joystick *stick2;             // a pointer to an uncreated Joystick object
stick2 = new Joystick(1);     // creates the instance of the Joystick object
stick2->GetX();                // access the instance with the arrow (->) operator
```

# Pointers and addresses

```
bot->ArcadeDrive(stick2);    // you can pass the instance by the pointer (notice, no &)
                             //      ...   ArcadeDrive(Joystick* j);    ...
delete stick2;              // delete it when your done with it
```

In the first group of statements (1) a Joystick object is created as a reference - stick1 refers to the object itself. In the second group of statements, stick2 is a pointer to a Joystick object.

[ArcadeDrive\(\)](#) in WPILib takes advantage of a C++ feature called *function overloading*. This allows it to have two methods with the same name that differ by argument lists. The one [ArcadeDrive\(Joystick &j\)](#) takes the parameter [j](#) as a reference to a [Joystick](#) instance. You supply a [Joystick](#) and the compiler automatically passes a reference to that instance. The other [ArcadeDrive\(Joystick \\*j\)](#) takes the parameter [j](#) as a pointer to a [Joystick](#) instance. You supply a pointer to a [Joystick](#) instance. The cool thing is that the compiler figures out which [ArcadeDrive](#) to call. The library is built this way to support both the pointer style and the reference style.

If you had non-overloaded functions [Ref\(Joystick &j\)](#) and [Ptr\(Joystick \\*j\)](#), you could still call them if you use the right C++ operators: [Ref\(\\*stick2\)](#) and [Ptr\(&stick1\)](#). At run time, references and pointers are both passed as addresses to the instance. The difference is the source code syntax and details like allocation and deletion.