

Class 2 Handout

Table of Contents

Review.....	2
Note for Class 2 Handout.....	4
If Statements.....	5
Types of If Statements.....	6
Types of Conditions.....	7
A Simple Program with Conditions.....	8
Loops.....	16
While Loops.....	16

Review

Last time we output text to the console, received input from the user, and we also talked about class Objects. Whenever you look at something in the physical world, try to think of what would be an Object, its Functionality, and Attributes it may have!

With Syntax, remember that commands end with a **semi-colon ;** Commands are usually statements that are one-line long.

Functions, If Statements, and Loops will not have a semi-colon at the end.

Classes have a semi-colon at the end of the closing curly-brace.

Curly Braces { } surround a body of code that belong together. This could be a class definition, a function's code, or code to be carried out within an **if statement** or **loop**.

```
int main()                                class Cat
{                                          {
    // Code goes here                    // Object's Functions and
                                          // Attributes definitions go
                                          // here.
                                          };
}
```

When we come across square brackets [], these are used for array indices. An array is essentially a list of numbers, characters, strings, etc. We will cover them later.

Basic datatypes:

- **int** – Integers. Whole numbers, positive, negative, and 0.
- **float** – Floats. 0, Positive & Negative numbers with decimal points.
- **char** – A single character; it could be any symbol such as numbers, letters, !@#\$%, etc.
- **string** – A string of characters. If you were storing names, it would be in a string.
- **bool** – Booleans. These will only ever have two values: True or False. Handy for asking the program questions to choose behavior, like “Is Car in Drive?”

Comments:

// Is a single-line comment

/* Lets you do

Multi-line Comments */

Parenthesis () are used to pass **variables** between functions, and they will also contain **conditional statements** for if statements and loops. If these conditional statements are **true**, the if statement or loop will run.

```
1  #include <iostream>
2  using namespace std;
3
4  bool IsNumberLucky( int number1 )
5  {
6      if ( number1 == 7 )
7      {
8          return true;
9      }
10 }
11
12 int main()
13 {
14     int num;
15     cout << "Enter a number: ";
16     cin >> num;
17
18     bool isLucky = IsNumberLucky( num );
19
20     if ( isLucky )
21     {
22         cout << "Your number " << num << " IS lucky!" << endl;
23     }
24     else
25     {
26         cout << "Your number " << num << " IS NOT lucky!" << endl;
27     }
28
29     return 0;
30 }
```

Note for Class 2 Handout...:

As you read through the sample source code in this handout, try to recognize which items are Variables, which items are Objects, and which items are Functions. Here is a list of how you can recognize different things:

Variable

```
int number;  
number = 2;  
cout << number << endl;
```

A variable is pretty simple-looking. There are no parenthesis or dots, it is just a name by itself.

Function

```
DisplayErrorMessage( errorCode );
```

You can recognize a function with the parenthesis. The parenthesis may be empty or contain **variables**, but a function always needs its parenthesis, whether being defined, or being called like in this example.

Class

```
cout << myAccount.balance << endl;  
myAccount.DepositMoney( 3.50 );
```

A class can hold **variables** and it can hold **functions**.

When you see an instance of a class being used, you will notice a period after its name, and then either a **variable name** or a **function** afterwards.

Here, we're outputting the account balance **variable**, and then we're calling the **DepositMoney** function and passing in a hard-coded value of \$3.50.

If Statements

Changing the way a program runs, based on variable state.

Last class, our programs would run from beginning to end, with no change. We were just inputting and outputting data.

With every-day computer software, there are many variables with different values, and the programs will behave differently based on an event, or a state.

If the user clicks "Save", open up the Save Dialog box

If the user clicks "Send Form", and the "Name" field is blank, open up Error Message

If the user presses the "Left Arrow" key, move the player avatar Left at velocity V



If Player and Bug collide, remove 10 health from Player's HP

And so on.

There could also be multiple outcomes, and a "default" case if none of the things we are looking for are true...:

```
if ( pizza.type == "Cheese" )  
{  
    cout << "Cheese is OK" << endl;  
}  
else if ( pizza.type == "Pineapple" )  
{  
    cout << "Pineapple is great!" << endl;  
}  
else  
{  
    cout << "I don't like this flavor" << endl;  
}
```

Here, we have an object **pizza**, which has an attribute **type**. For this scenario, the **type** variable is a **string**.

Types of If Statements

In C++, you can have an if statement by itself:

```
cin >> guessedNumber;  
if ( guessedNumber == secretNumber )  
{  
    cout << "You Win!" << endl;  
    return 0;  
}  
// Keep playing
```

You can consider whether the statement is **true** or **false** with an if/else statement:

```
if ( number1 == number2 )  
{  
    cout << "Number 1 and Number 2 are equal" << endl;  
}  
else  
{  
    cout << "Number 1 and Number 2 are not equal" << endl;  
}
```

And you can also through an **else if** in the works.

```
if ( operatingSystem == "Windows" )  
{  
    LoadWindowsSettings();  
}  
else if ( operatingSystem == "Linux" )  
{  
    LoadLinuxSettings();  
}  
else  
{  
    cout << "Sorry, your OS is not supported" << endl;  
}
```

You can have as many **else if** statements as you want, and you don't need to have an **else** statement at all if you don't want to handle that state.

Types of Conditions

You can ask different types of questions in C++. Above, we check to see if things are equal to each other, but we can also ask if numbers are greater than or less than, if something is *not* another thing, or if a boolean statement is *true* or *false*.

```
if ( price == 19.99 )           // is equal to
if ( accountBalance != 0 )      // is not equal to
if ( player.y > 100 )           // greater than
if ( player.y >= 100 )          // grater than or equal to
if ( accountBalance < 0 )       // less than
if ( velocity <= 0 )           // less than or equal to
if ( doneWithProgram == true ) // if statement is true
if ( doneWithProgram )         // if statement is true (short-hand)
if ( available == false )      // if statement is false
if ( !available )              // if statement is false (short-hand)
```

Notice that when we're asking if something is equal to another, we use **double-equals** ==. However, if we're assigning a value to a variable, we use **single-equals** =.

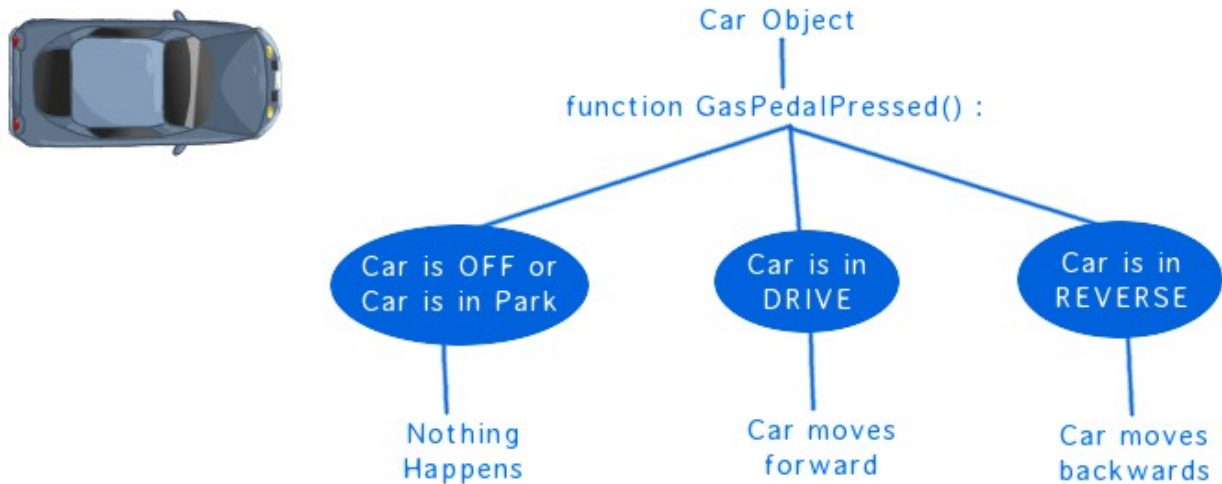
Additionally, you can also string conditionals together with "and" and "or".

"and" shows up as &&, and "or" shows up as || (this is two pipes – press shift and \)

```
/* AND STATEMENT */
if ( player.alive == true && player.velocity > 0 )
{
    // If player is alive and their velocity is greater than 0
    player.MoveForward();
}

/* OR STATEMENT */
if ( fileMenu.SaveButtonHit() || keyboardShortcut.SaveKeyHit() )
{
    // if the Save button was hit out of the file menu,
    // OR if the Save keyboard shortcut was hit
    textDocument.SaveChanges();
}
```

So, now when you're looking at **objects** around the room and their **functionality**, try to also take into consideration how it might act different, **if one thing is true, or another thing is**.



How about a TV Remote? What functionality does it have, and how does it behave based on the state of the Television?

A Simple Program with Conditions

So, let's create a "Virtual Car" program.

With our virtual car, we can start the car, change gears, and hit the gas. The logic of how it behaves is outlined in the tree above.

So, let's create a new project by going to

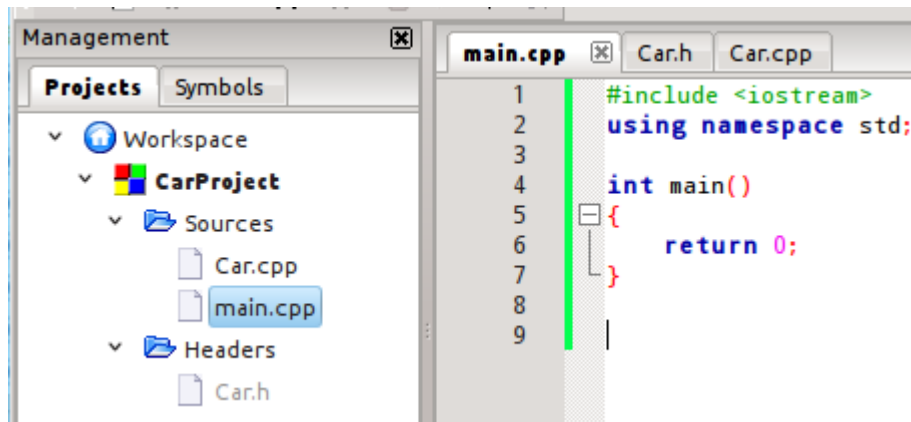
File > New > Project...

And then create new files with

File > New > Empty File

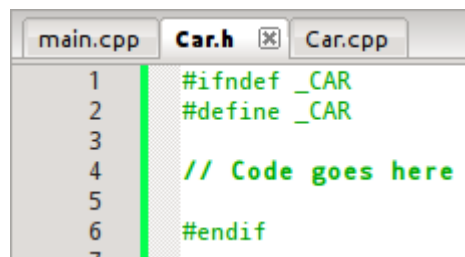
We will want to create **main.cpp**, **Car.h**, and **Car.cpp**.

We will start by adding the bare-basics of what we need in **main**:



Again, *main()* is where the program starts from, so we always need to define this in some file. Our file doesn't *need* to be **main.cpp**, it could be **class2.cpp** or anything else you want, but we need *int main()* defined in the program.

In Car.h, we will create our class **prototype**. In C++, it is standard for a class to have its own source files. The prototype (or declaration) goes in the **header .h** file, and the definition goes in the **source .cpp** file.



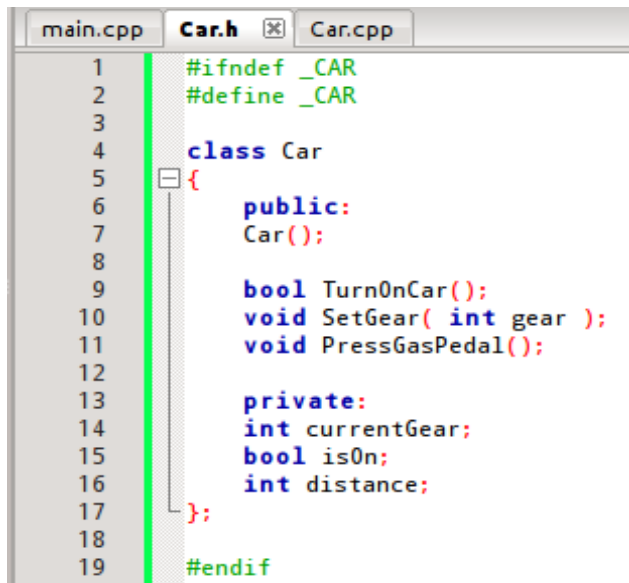
In our header file, we need to start by adding:

```
#ifndef _CAR
#define _CAR

#endif
```

These are preprocessor commands, that essentially tell the compiler *if you've read this file already, skip it*. If we didn't have these commands, and we were including the Car file in multiple source files (which is not uncommon), it would read this file multiple times, and think we were declaring our Car class over and over!

Literally, these preprocessors are saying "if `_CAR` is not defined (ifndef), define `_CAR`". Therefore, if `_CAR` has been defined, it won't go over the code a second time.



```
1  #ifndef _CAR
2  #define _CAR
3
4  class Car
5  {
6      public:
7      Car();
8
9      bool TurnOnCar();
10     void SetGear( int gear );
11     void PressGasPedal();
12
13     private:
14     int currentGear;
15     bool isOn;
16     int distance;
17 };
18
19 #endif
```

When we create our class, we will have private variables like **currentGear**, **isOn**, and **distance**.

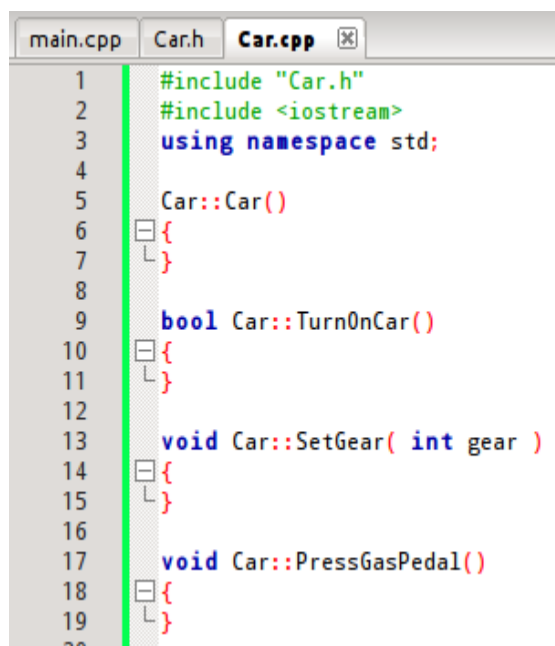
These variables are hidden from the user, and only the car needs to know.

Then we have our public members. This is what the driver will interface with.

The driver will be able to **TurnOnCar**, **SetGear**, and **PressGasPedal**.

Finally, there is **Car()**; defined on line 7. This is a **Constructor**, and you can think of it like a function that executes immediately after we create a new car.

Next, lets open the **Car.cpp** file:



```
1  #include "Car.h"
2  #include <iostream>
3  using namespace std;
4
5  Car::Car()
6  {
7  }
8
9  bool Car::TurnOnCar()
10 {
11 }
12
13 void Car::SetGear( int gear )
14 {
15 }
16
17 void Car::PressGasPedal()
18 {
19 }
20
```

We don't need the `#ifndef` preprocessor commands here, since this is a source file. This is where we set what all the functions actually do.

On lines 1 and 2, we're including **Car.h**, and **iostream**. We need to include the corresponding .h file for this .cpp file.

Then, we have all of the functions from Car.h listed, but with curly-braces to hold our function bodies.

Notice that on line 5, the Constructor **Car()** has no return-type. Constructors are meant for initializing variables, and do not return data.

Also notice that each of the function names are prefixed with **Car::**. When we write functions outside of the class itself, we need to specify which class it belongs to. We do this with the **Car::** prefix.

However, if we call one of these functions (ie, from main) we do not need that prefix.

So in our class, we have three variables. Lets initialize what they are:

```
Car::Car()  
{  
    distance = 0;  
    isOn = false;  
    currentGear = 0;  
}
```

With **distance**, we're going to keep track of how far forward and backward the car goes. We're starting it out at point "0".

We're starting the car off, by setting the **isOn** boolean variable to **false**.

And we're setting the current gear to 0. We can say that 0 is the code for being "In Park".

Then, we can define what happens when
the user tries to turn on the car:

```
bool Car::TurnOnCar()  
{  
    if ( isOn == false )  
    {  
        isOn = true;  
        return true;  
    }  
    return false;  
}
```

Here, we are first asking, "Is the car off?" by saying:
If the variable **isOn** is **false**...

If the car is currently off, we turn it on by setting
isOn = true;

and then we **return true;** because the car turned on
successfully.

When we **return** a value inside of a function, nothing else in that function is executed. Once we **return true** from within the if statement, it will never reach the **return false** statement.

What we have above is essentially an "if/else", but it would be redundant to have an "else" in this case, since if the car is on then there's nothing else TO return but false!

```
void Car::SetGear( int gear )  
{  
    // 0 = Park, 1 = Drive, 2 = Reverse  
    if ( gear >= 0 && gear < 3 )  
    {  
        currentGear = gear;  
    }  
}
```

For **SetGear**, we are allowing the user to pass
in a value for "Gear".

We are only going to accept 0, 1, and 2 as valid
gears for the car to be in, so we need to check
that what was passed in was valid, first.

With our if statement, we are asking,

Is the gear between 0 and 3?

by asking

**If the gear is greater-than-or-equal-to 0, AND
if the gear is also less than 3**

If the gear is a valid number, we change our currentGear. Otherwise, we ignore the command.

And when we're handling the gas pedal being pressed, we have to consider the state of the car -- Is the car on? Is it in drive? And so on.

```
31 void Car::PressGasPedal()
32 {
33     if ( isOn == false || currentGear == 0 )
34     {
35         // Either the car is not on, or we're in park.
36         cout << "You press the gas pedal, but the car does not move." << endl;
37     }
38     else if ( currentGear == 1 )
39     {
40         // We are in drive
41         cout << "You drive the car forward" << endl;
42         distance += 1;
43     }
44     else if ( currentGear == 2 )
45     {
46         // We are in reverse
47         cout << "You drive the car backward" << endl;
48         distance -= 1;
49     }
50
51     cout << "Distance is " << distance << endl;
52 }
```

On the first **if** statement, we check to see if the car is off, OR if the car is in Park. If one or both of these are true, then nothing happens and we output a message about the car not moving.

Because this first **if** handles the case of the car being off, for the subsequent **else if** statements, it is not possible for the car to be off and handled by these.

If the car is in drive, then we will output that the user has moved forward, and we will add 1 to the distance.

We can increment the distance in different ways:

```
distance = distance + 1;
distance += 1;
distance++;
++distance;
```

Each of those commands will allow us to add one to the distance. The top two will let us increment by some number other than 1, as well.

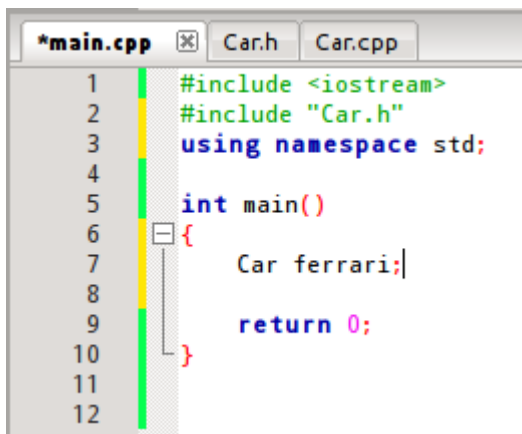
If the car is in reverse, we merely subtract the distance.

And, finally, whether the car stays in the same place or moves, we are going to output the current distance.

So we have defined what a Car is, and how it behaves. We have created the **object**, its **attributes**, and its **functionality**.

Now that we have a Car, we need some way to interface with it.

Let's go back to **main.cpp**, and add a Car.:



```
1 #include <iostream>
2 #include "Car.h"
3 using namespace std;
4
5 int main()
6 {
7     Car ferrari;
8
9     return 0;
10 }
11
12
```

We have added the include on line 2, and a new Car on line 7.

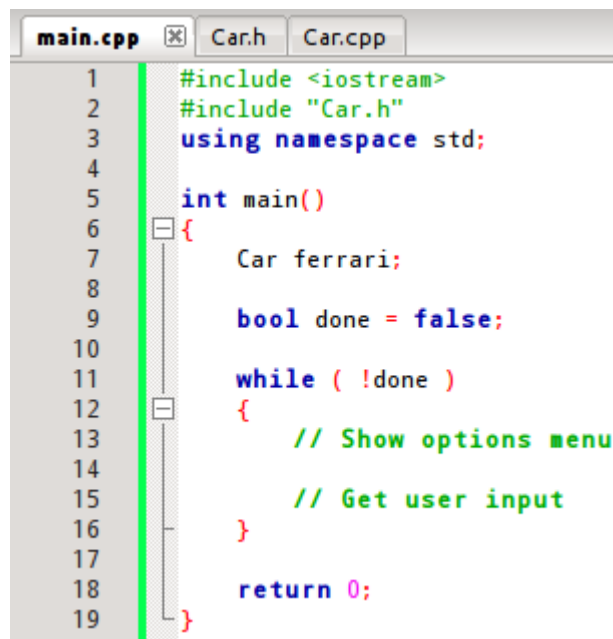
We do not need to include the .cpp file, only the .h file.

To have the user interface with the car, we will want to get input from the user.

Additionally, we don't want the program to quit after just one runthrough, we'd like to see the distance get larger or smaller as the player drives!

Well, let's throw in a loop, even though we haven't covered what they are yet. We will essentially be adding a condition:

While the player still wants to play, run the program again!



```
1 #include <iostream>
2 #include "Car.h"
3 using namespace std;
4
5 int main()
6 {
7     Car ferrari;
8
9     bool done = false;
10
11     while ( !done )
12     {
13         // Show options menu
14         // Get user input
15     }
16
17
18     return 0;
19 }
```

So, we're creating a **bool** value to store whether we're done with the program. Then, while the program is still running, we will do things like show the user a menu each time, and then get their input.

Our menu might look something like:

```
Choose a command:
 1. Turn on car
 2. Change gear
 3. Press gas pedal
 4. Quit program
```

And only when the user selects option 4, will we change **done** to **true**.

```
11 while ( !done )
12 {
13     // Show options menu
14     cout << "CHOOSE AN OPTION" << endl;
15     cout << "\t 1. Turn on car" << endl;
16     cout << "\t 2. Change gear" << endl;
17     cout << "\t 3. Press gas pedal" << endl;
18     cout << "\t 4. Quit" << endl;
19
20     // Get user input
21     int choice;
22     cin >> choice;
23 }
```

Inside the **while** loop, output menu options and then get the user's input.

Once we get the user's choice, we will have another if statement to figure out what to do:

```
while ( !done )
{
    // Show options menu
    cout << "CHOOSE AN OPTION" << endl;
    cout << "\t 1. Turn on car" << endl;
    cout << "\t 2. Change gear" << endl;
    cout << "\t 3. Press gas pedal" << endl;
    cout << "\t 4. Quit" << endl;

    // Get user input
    int choice;
    cin >> choice;

    if ( choice == 1 )
    {
        ferrari.TurnOnCar();
    }
}
```

```
        else if ( choice == 2 )
        {
            cout << "What gear? ";
            cin >> choice;
            ferrari.SetGear( choice );
        }
        else if ( choice == 3 )
        {
            ferrari.PressGasPedal();
        }
        else if ( choice == 4 )
        {
            done = true;
        }
        else
        {
            cout << "Invaild command!" << endl;
        }
    }
```

Notice that we even have an "else", for when the user inputs something besides options 1 – 4. It will just let them know the command was invalid, and then the loop will repeat to get their next command.

So now if we run the "game", we will get something like:

```

4. Quit
3
You press the gas pedal, but the car does not move.
Distance is 0
CHOOSE AN OPTION
1. Turn on car
2. Change gear
3. Press gas pedal
4. Quit
2
What gear? 1
CHOOSE AN OPTION
1. Turn on car
2. Change gear
3. Press gas pedal
4. Quit
3
You drive the car forward
Distance is 1
CHOOSE AN OPTION
1. Turn on car
2. Change gear
3. Press gas pedal
4. Quit
■
```

If you want the entire sample code for this, go to the "Samples" folder under "Class 02" on the class website.

Not the prettiest thing ever, and could use some cleaning up, but I'll leave that to you to figure out. :)

Loops

I know you're feeling disappointed that I used a while loop in our Car program. It's almost like I spoiled what your Christmas present was going to be. I'm sorry. But don't worry, there are more to loops than just that!

While Loops

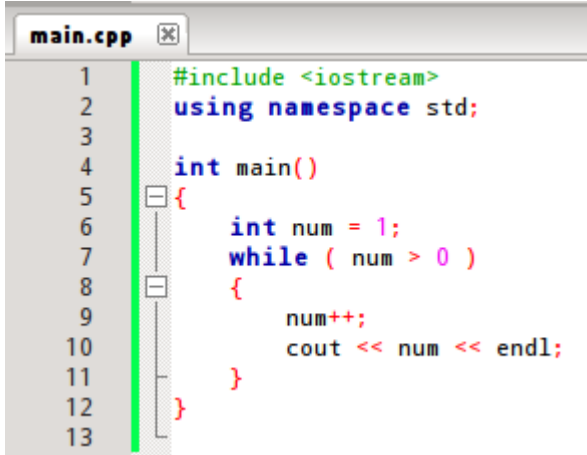
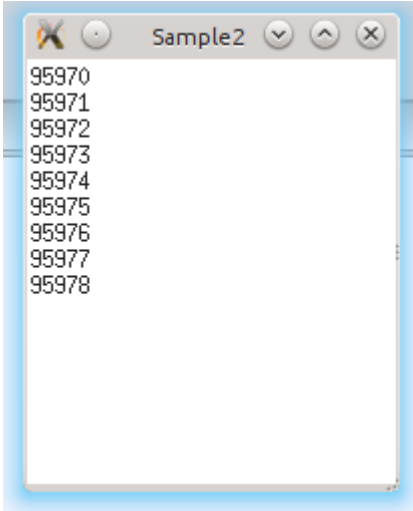
While loops look really similar to if statements:

```
if ( lunchToday == "creamed corn" )  
{  
}
```

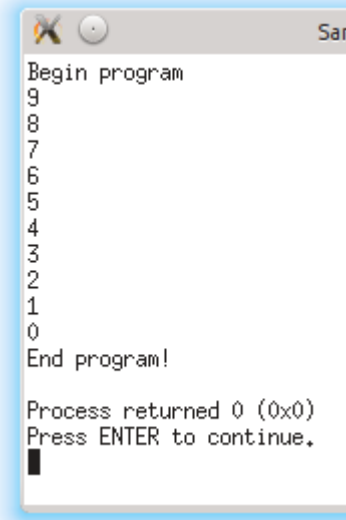
```
while ( lunchToday != "chicken nuggets" )  
{  
}
```

They both have a conditional statement within parenthesis (...), and then have a body following, for code to run if that conditional statement is true.

The difference, however, is that the loop will run multiple times, up until the conditional statement becomes **false**. Because of this, it is easy to get trapped in an infinite loop:

 <pre>main.cpp 1 #include <iostream> 2 using namespace std; 3 4 int main() 5 { 6 int num = 1; 7 while (num > 0) 8 { 9 num++; 10 cout << num << endl; 11 } 12 } 13</pre>	 <pre>Sample2 95970 95971 95972 95973 95974 95975 95976 95977 95978</pre> <p><i>Are we there yet?</i></p>
---	---

The program relies on the fact that something inside the while loop really ought to make the while's conditional statement false.

<pre>1 #include <iostream> 2 using namespace std; 3 4 int main() 5 { 6 cout << "Begin program" << endl; 7 int num = 10; 8 while (num > 0) 9 { 10 num--; 11 cout << num << endl; 12 } 13 cout << "End program!" << endl; 14 } 15</pre>	
--	--

While loops could be useful for:

- Continuing a game until the user hits quit
- Running through a list of items and processing them
- Looping until the user finally enters a valid option

And other things.

We'll go over this more next class. This was just a brief introduction to loops!