COMP 1406: Introduction to C++

C++ Basics

```
#include <iostream>

int main(int argc, char **argv)
{
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

```
#include <iostream>
```

Includes a system header file – this gives you access to system functions like cout

```
int main(int argc, char **argv)
{
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

```
#include <iostream>

int main(int argc, char **argv)
{
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

The `main()` function is the entry point of the program

```
#include <iostream>

int main(int argc, char **argv)
{
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

The C++ equivalent of `print()`, `println()`, or `printf()`

```
#include <iostream>

int main(int argc, char **argv)
{
    std::cout <<                                    endl;
    return 0;
}
```

Returning 0 when the program is finished lets whoever started the program know that there were no errors.

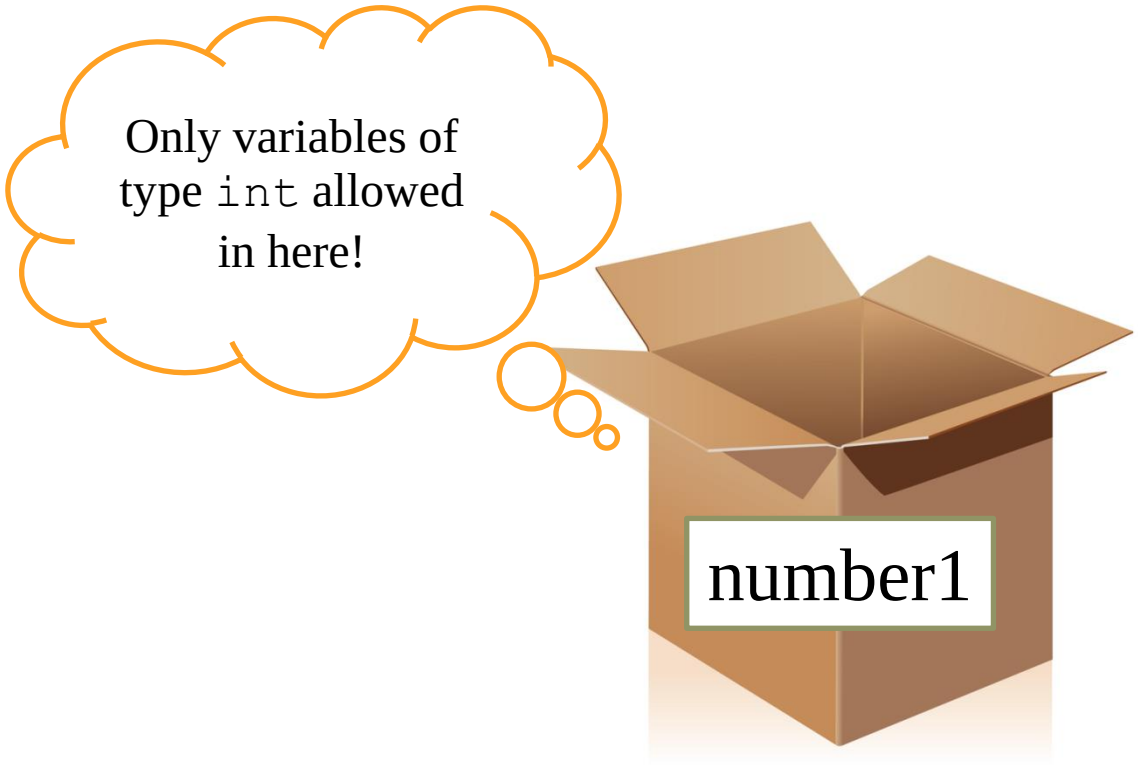
```
#include <iostream>

int main(int argc, char **argv)
{
    std::cout << "Hello World" << std::endl;

}
```

We can eliminate needing this by adding a new line at the top of the file…

```
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    cout << "Hello World" << endl;
    return 0;
}
```

Variables and Data Types

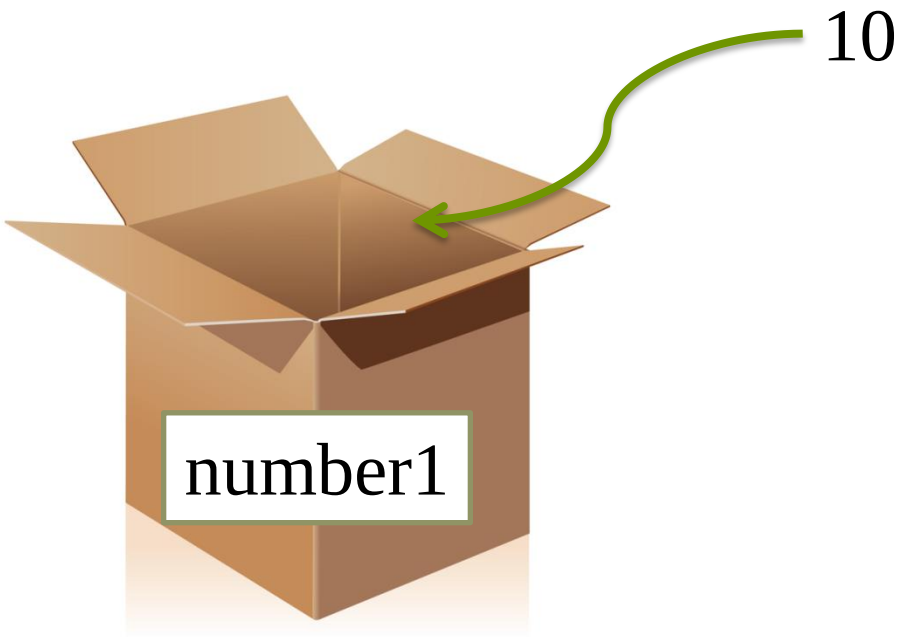
```
#include <iostream>

using namespace std;

int main()
{
    int number1;
    float number2 = 1.4f;

    number1 = 10;

    cout << number1 << ", " << number2 << endl;

    return 0;
}
```



Variable type → `int number1;`

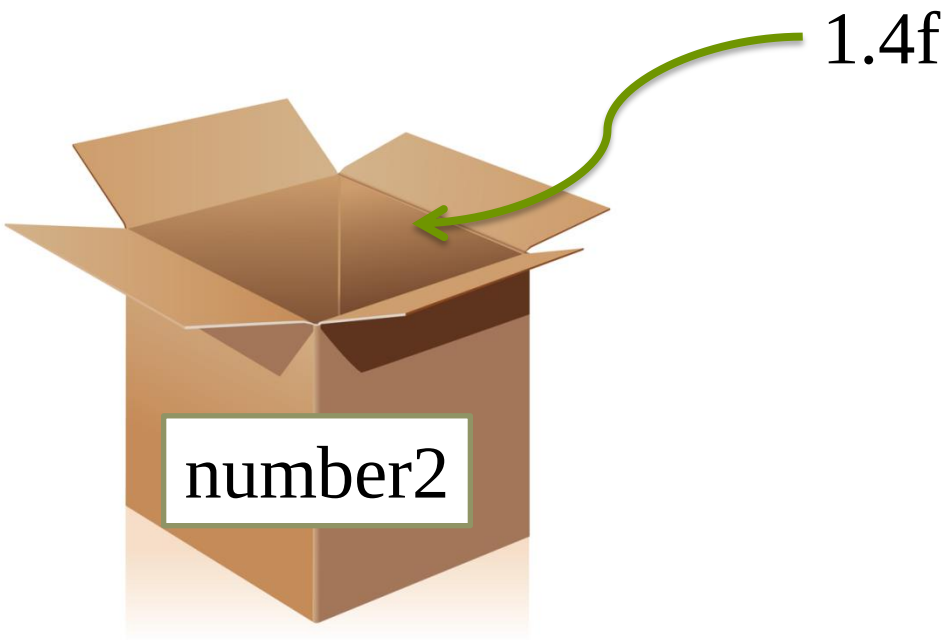
Variable declaration!

Data Type	Info About Type
`int`	whole numbers
`char`	single characters
`bool`	Boolean: true or false
`float`	floating point numbers with a fractional part
`double`	higher precision numbers with a fractional part
`string`	a string of characters (need to `#include <string>`)



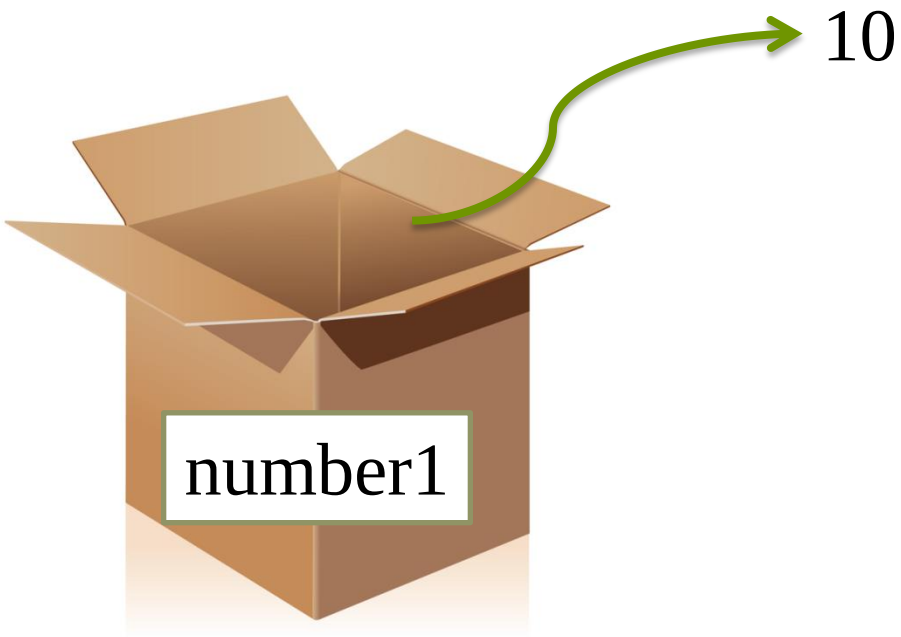

```
number1 = 10;
```

Variable assignment!




```
float number2 = 1.4f
```

Variable declaration AND assignment!




```
cout << number1 << ", " << number2 << endl;
```

Using the variable's value

Memory Address	Data Stored
500	0b11011010
501	0b11101010
502	0b00000000
503	0b00000000
504	0b00010011
505	0b11111111
506	0b01000111

Memory Addr	
500	
501	
502	0b00000000
503	0b00000000
504	0b00010011
505	0b11111111
506	0b01000111

Each location in memory represents one byte (8 bits)

Memory Address	Data Stored
500	0b11011010
501	
502	
503	
504	0b00010011
505	0b11111111
506	0b01000111

This is a binary number. Since it has 8 digits (i.e. bits), it represents one byte.

Memory Address	Identifier	Data Stored
500	number1	
501		
502		
503		
504		
505		
506		


```
int number1;
```

Declaring a variable saves the number of bytes it will need in memory, determined by data type.

Memory Address	Identifier	Data Stored
500	number1	10
501		
502		
503		
504		
505		
506		


```
number1 = 10;
```

Assigning data to a variable puts the data in the reserved space.

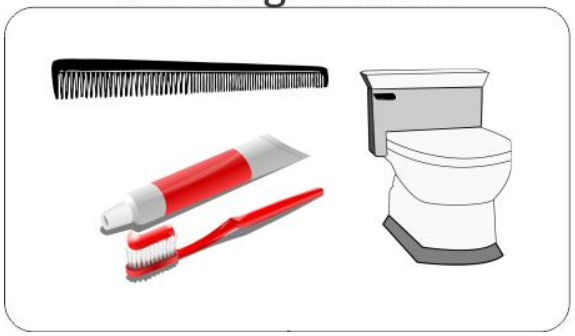
Functions

Functions as Routines

```
void printNumbers()
{
    cout << "1, 2, 3, 4, 5, 6" << endl;
}


int main()
{
    printNumbers();
    return 0;
}
```

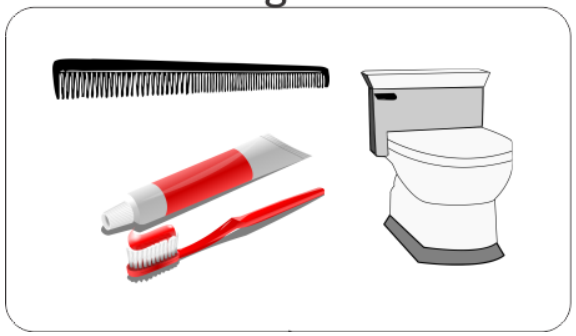
Functions as Routines

morningRoutine

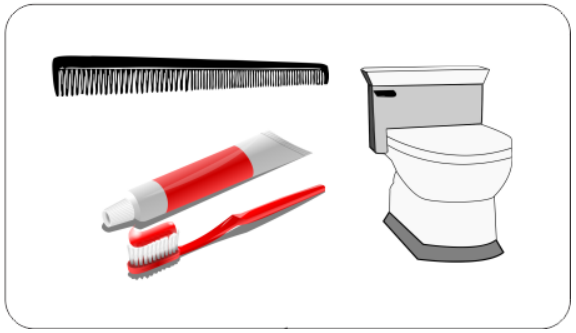


Functions as Routines

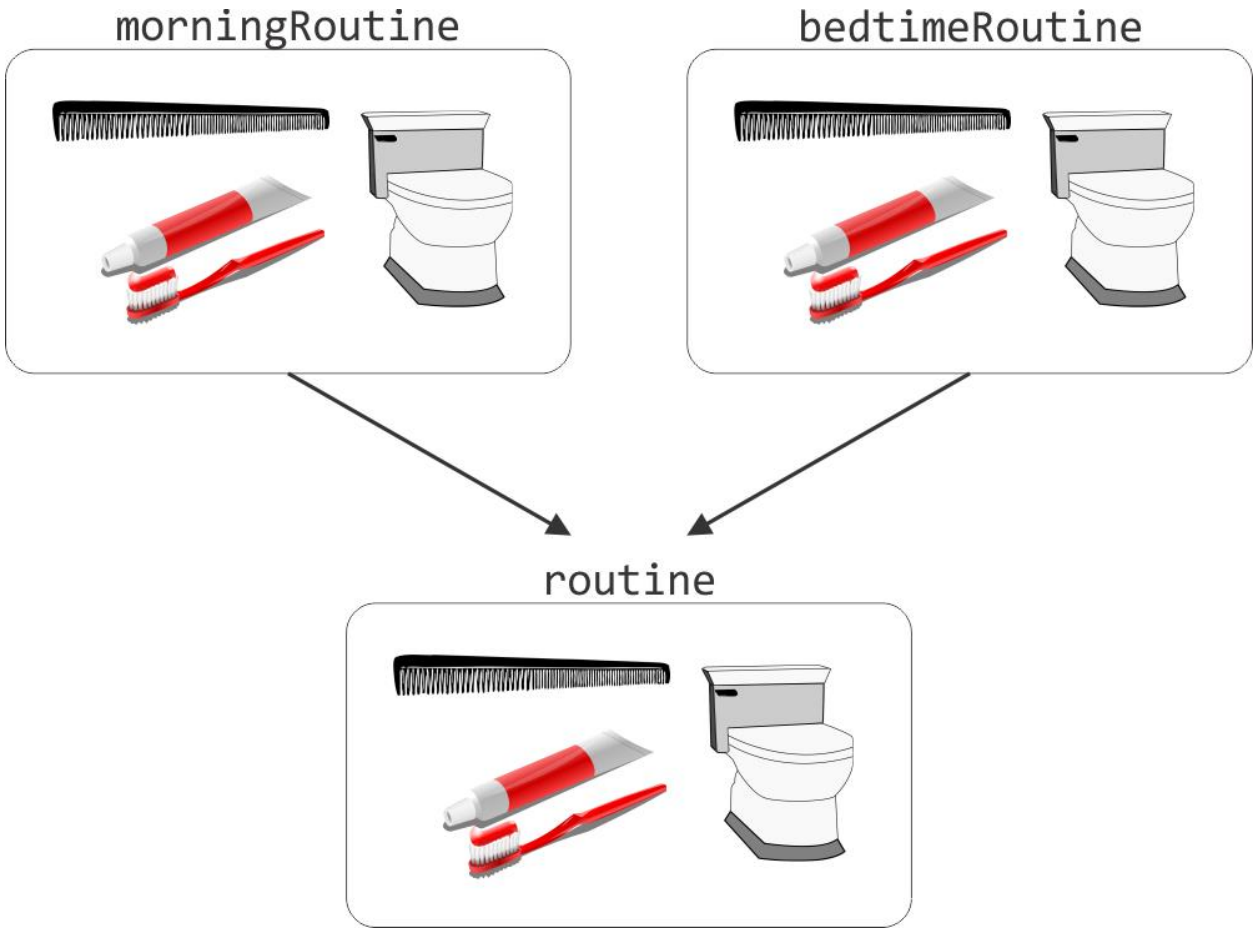
`morningRoutine`



`bedtimeRoutine`



Functions as Routines



Functions as Routines

```
void printNumbers()
{
    c                , 3, 4, 5, 6" << endl;
}


int main()
{
    printNumbers();
    return 0;
}
```

Function name

Functions as Routines

```
void printNumbers()
{
    cout <<                                  6" << endl;
}


int main()
{
    printNumbers();
    return 0;
}
```

Parameter list (empty in this case)

Functions as Routines

```
void printNumbers()
{
    cout << "1, 2, 3, 4, 5, 6" << endl;
}

int main()
{
    printNumbers();
    return 0;
}
```

Return type (nothing is returned in this case)

Functions as Routines

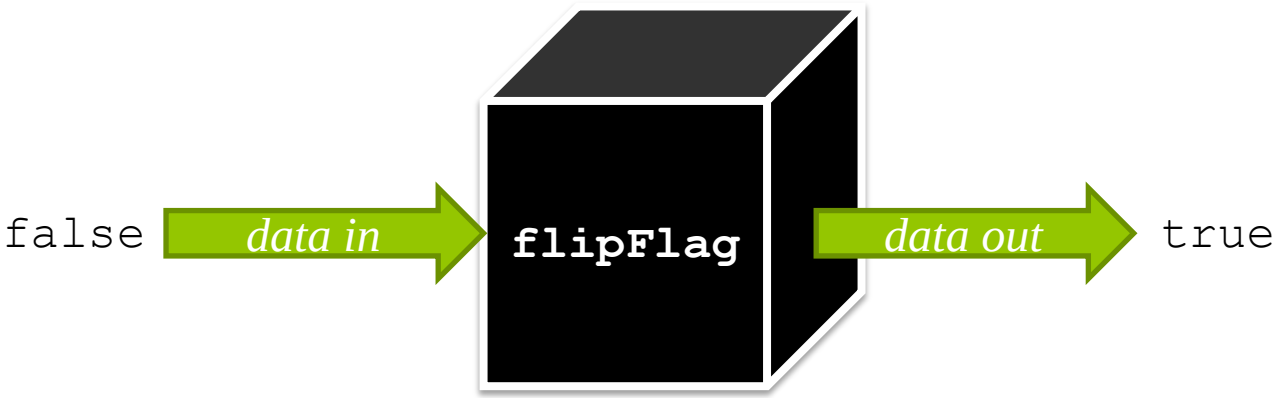
```
void printNumbers()
{
    cout << "1, 2, 3, 4, 5, 6" << endl;
}


int main()
{
    printNumbers();
    return 0;
}
```

Calling the function ("invoking the routine")

Functions As a Black Box

```
bool flipFlag(bool flag)
{
    return !flag;
}

int main()
{
    bool flag = false;

    cout << flag << endl;
    flag = flipFlag(flag);
    cout << flag << endl;

    return 0;
}
```

Functions As a Black Box



The user of flipFlag doesn't care how the operation is done!

Functions As a Black Box

```
bool flipFlag(bool flag)
{
    return
}

int main()
{
    bool flag = false;

    cout << flag << endl;
    flag = flipFlag(flag);
    cout << flag << endl;

    return 0;
}
```

Single parameter of type `bool` (data in)

Functions As a Black Box

```
bool flipFlag(bool flag)
{
    return !flag;
}

int main()
{
    bool flag = false;

    cout << flag << endl;
    flag = flipFlag(flag);
    cout << flag << endl;

    return 0;
}
```

Return type: a promise to return a value of type `bool`

Functions As a Black Box

```
bool flipFlag(bool flag)
{
    return !flag;
}

int
{
    bool flag = false;

    cout << flag << endl;
    flag = flipFlag(flag);
    cout << flag << endl;

    return 0;
}
```

Returning a value
(data out)

Functions As a Black Box

```
bool flipFlag(bool flag)
{
    return !flag;
}

int main()
{
    bool flag = false;

    cout << flag << endl;
    flag = flipFlag(flag);
    cout << flag << endl;

    return 0;
}
```

Notice when we call `flipFlag`, we do not care how it is implemented

Functions As a Black Box

```
bool flipFlag(bool flag)
{
    return !flag;
}

int main()
{
    bool flag = false;

    cout << flag << endl;
    flag = flipFlag(flag);
    cout << flag << endl;
}
```

Saving the value that was returned from the function

Conditionals and Switch

Single If-Statement

```
int main()
{
    if (5 < 10)
    {
        cout << "Option 1" << endl;
    }

    return 0;
}
```

Single If-Statement

Boolean expression: can be anything that results in true or false

```
in
{
     if (5 < 10)
     {
          cout << "Option 1" << endl;
     }

     return 0;
}
```

Note: zero is false and any non-zero number is true in C++!

Single If-Statement

```
int main()
{
    if (5 < 10)
    {
        cout << "Option 1" << endl;
    }

    return 0;
}
```

The block of statements that are executed when the Boolean expression is true

Note: if there are no braces, then the next line after the "`if ()`" makes up the block, regardless of indentation!

If-Else Statement

```
int main()
{
    if (5 < 10)
    {
        cout << "Option 1" << endl;
    }
    else
    {
        cout << "Option 2" << endl;
    }
    return 0;
}
```

If-Else Statement

```
int main()
{
    if (5 < 10)
    {
        cout << "Option 1" << endl;
    }
    else
    {
        cout << "Option 2" << endl;
    }
    return 0;
}
```

The block of statements that are executed when the Boolean expression is false

If / Else-If / Else

```
int main()
{
    if (5 < 10)
    {
        cout << "Option 1" << endl;
    }
    else if (3 < 4)
    {
        cout << "Option 2" << endl;
    }
    else
    {
        cout << "Option 3" << endl;
    }
    return 0;
}
```

Start at the beginning of a chain, and stop at the first true Boolean expression (or just run the `else` if there is one and nothing else was true)

Switch Statement

```
int main()
{
    int x = 2;
    switch (x)
    {
        case 1:
            cout << "Option 1" << endl;
            break;
        case 2:
            cout << "Option 2" << endl;
            break;
        default:
            cout << "Default" << endl;
            break;
    }

    return 0;
}
```

Switch Statement

```
int main()
{
    int x = 2;
    switch (x)
    {
        case 1:
            cout << "Option 1" << endl;
            break;
        case 2:
            cout << "Option 2" << endl;
            break;
        default:
            cout << "Default" << endl;
            break;
    }

    return 0;
}
```

The value you want to find a match for

Switch Statement

```
int main()
{
    int x = 2;
    switch (x)
    {
        case 1:
            cout << "Option 1" << endl;
            break;
        case 2:
            cout << "Option 2" << endl;
            break;
        default:
            cout << "Default" << endl;
            break;
    }

    return 0;
}
```

Each case represents the statements that should be executed if the switched value matches

Switch Statement

```
int main()
{
    int x = 2;
    switch (x)
    {
        case 1:
            cout << "Option 1" << endl;
            break;
        case 2:
            cout << "Option 2" << endl;
            break;
        default:
            cout << "Default" << endl;
            break;
    }

    return 0;
}
```

The value that needs to match with the switched value for this case to be selected

Switch Statement

```
int main()
{
    int x = 2;
    switch (x)
    {
        case 1:
            cout <<
            break;
        case 2:
            cout <<
            break;
        default:
            cout << "Default" << endl;
            break;
    }

    return 0;
}
```

Without a `break`, the statements in the next case will continue to execute even if this case was selected (this is sometimes desirable)

Switch Statement

```
int main()
{
    int x = 2;
    switch (x)
    {
        case 1:
            cout << "Option 1" << endl;
            break;
        case 2:
            cout << "Option 2" << endl;
            break;
        default:
            cout <
            break;
    }

    return 0;
}
```

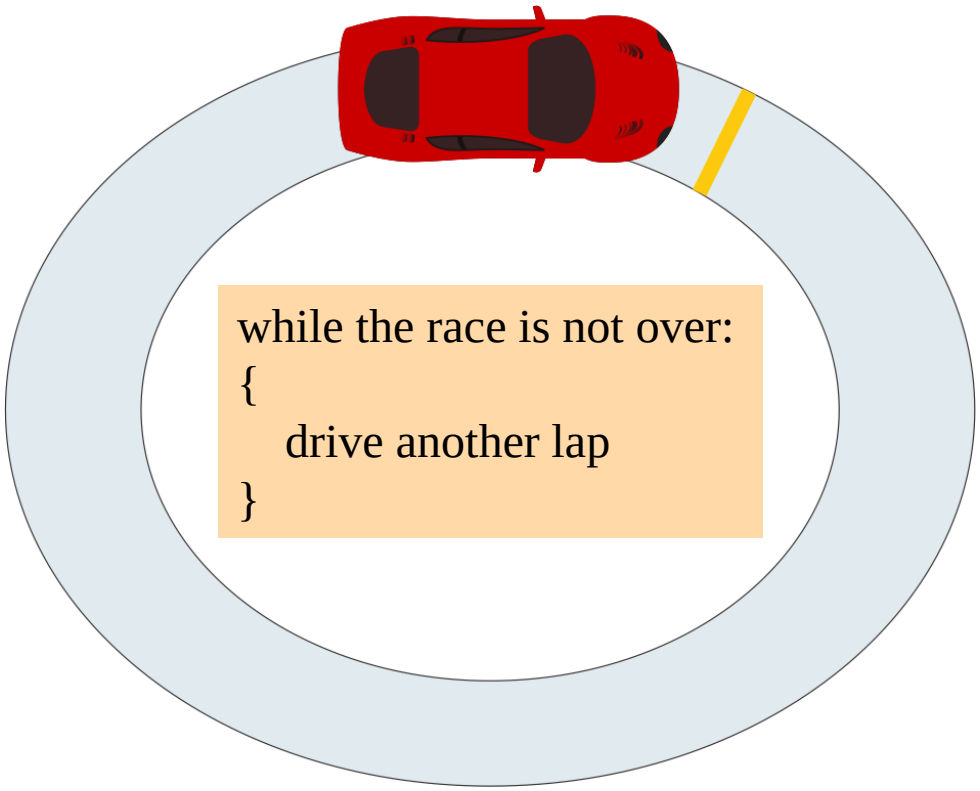
The default case is executed when nothing else matched

Iteration

While Loops

```
int main()
{
    int x = 0;
    while (x < 10)
    {
        x++;
        cout << x << endl;
    }

    return 0;
}
```

While Loops



While Loops

```
int main()
{
    int x = 0;
    while (x < 10)
    {
        x++;
        cout << x << endl;
    }

    return 0;
}
```

The Boolean expression that gets checked before the block of statements is executed again

While Loops

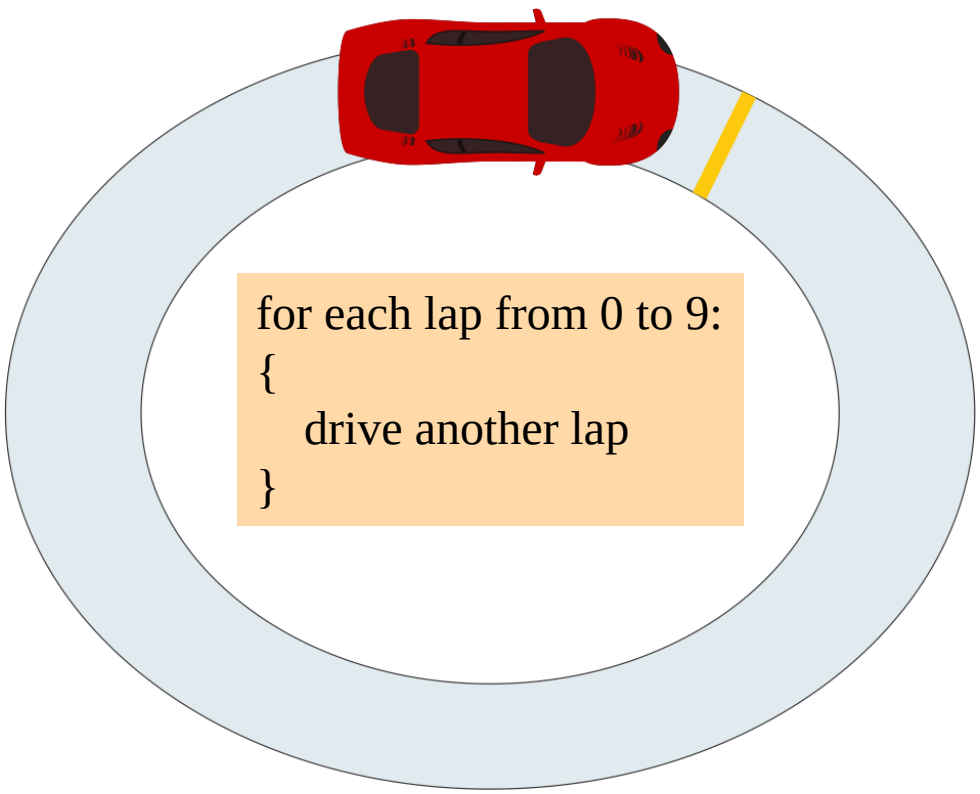
```
int main()
{
    int x = 0;
    while (x < 10)
    {
        x++;
        cout << x << endl;
    }

    return 0;
}
```

The entire block of statements is run if the Boolean expression is true (just like an if-statement)

For Loops

```
int main()
{
    for (int x=0; x < 10; x++)
    {
        cout << x << endl;
    }

    return 0;
}
```

For Loops



For Loops

```
int main()
{
    for (int x=0; x < 10; x++)
    {
        cout << x << endl;
    }

    return 0;
}
```

First, the initialization statement is executed…

For Loops

…then the condition is checked (if it doesn't pass, the loop ends)…

```
int main()
{
    for (int x=0; x < 10; x++)
    {
        cout << x << endl;
    }

    return 0;
}
```

For Loops

```
int main()
{
    for (int x=0; x < 10; x++)
    {
        cout << x << endl;
    }

    return 0;
}
```

…then the block of statements is executed…

For Loops

```
int main()
{
    for (int x=0; x < 10; x++)
    {
        cout << x << endl;
    }

    return 0;
}
```

…then the update statement is executed before checking the condition again.

While vs. For Loops

When should we use while loops?

When should we use for loops?

Putting it All Together

Poll Everywhere Question

What will the following loop output?

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Text 37607

217213
6, 6
5, 5
4, 4
3, 3
2, 2
1, 1

217215
7, 6
5, 4
3, 2
1, 0

219188
6, 5
4, 3
2, 1

224450
5, 4
3, 2
1, 0

224451
The loop will be infinite.

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Expression:

Anything that results in a value.

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Literal expression

Expression:

Anything that results in a value.

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Boolean expression

Expression:

Anything that results in a value.

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Statement:

A line of code that ends with a semi-colon. They are executed one by one in the order they appear unless flow control statements are used.

```
int addOne(int num)
{
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

return statement

Statement:

A line of code that ends with a semi-colon. They are executed one by one in the order they appear unless flow control statements are used.

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Block:

A collection of statements within curly braces: `{` and `}`.

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Block:

A collection of statements within curly braces: `{` and `}`.

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Block:

A collection of statements within curly braces: `{` and `}`.

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Block:

A collection of statements within curly braces: `{` and `}`.

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Control Flow:

In what order are statements executed?

```
int addOne(int num)
{
    num++;
    return num;
}

int main()
{
    int y = 5;
    while (y >= 0)
    {
        int x = addOne(y);
        cout << x << ", " << y;
        cout << endl;
        y -= 2;
    }
    return 0;
}
```

Data Flow:

What variables are in memory, and how do their values change over time?