Flow Control and Functions

Chantilly Robotics (Team 612)

Flow Control

An essential skill

Flow control statements

- Controls order of execution of code
- Looping
  - `while`, `do-while`, `for`
- Conditional
  - `if`, `if-else`, `switch`
- Jump
  - `return`, `break`, `continue`

Note: if you want to try out the examples, you may need to `#include <iostream>` or `#include <string>`

`if` and `if-else` statement

Use to execute code once based on a condition

Condition is in parentheses

```
if(x > 10)
    std::cout << "x > 10";
```

Print statement only executed if condition is true

This statement executed if condition is true

```
if(x > 20)
    std::cout << "x > 20";
else
    std::cout << "x <= 20";
```

This statement executed if condition is false

`if-else` can also be stacked

“`if-else` ladder”

```
if(x < 0)
    std::cout << "x is negative";
else if(x > 0)
    std::cout << "x is positive";
else
    std::cout << "x is zero";
```

Use braces for multiple statements

```
if(x < 0) {
    x += 5;
    std::cout << "Added 5";
}
```

```
if(x > 0 && y > 0) {
    x *= -1;
    y *= -1;
} else {
    x -= 10;
    y = x * 7;
}
```

When adding additional statements to the body, remember to add the braces!

Also, `while`, `do-while`, and `for` loops use braces in this way

`switch` block

- Syntax:
  - `switch(expression) { switchbody }`
- Only used with `int` type
- Like a limited `if-else` ladder
- Body made up of `case` labels
  - Selected based on value of expression
  - Must be compile-time constants
  - `default:` label for default case
- `break` statements needed after every case
  - Otherwise fallthrough occurs
  - Sometimes useful

```
case 6:
case 7:
    /* code */
    break;
```

```
constexpr int SIX = 6;
/* constexpr */ int five = 5;

int n = /* initialize n */;
switch(n) {

    case 3: //ok, 3 is a constant
        std::cout << "three";
        break;
    case SIX: //ok, SIX is a constant
        std::cout << "six";
        break;
    case five: //error, five is not a constant
        std::cout << "five";
        break;
    default: //default case
        std::cout << "default";
}
```

Note: use `switch` sparingly; there are generally better ways to achieve this behavior.

`while` loop and `do-while` loop

Condition executed at the **start** of each iteration. If true, the body executes; if not, the loop ends.

```
while(x > 0) {
    std::cout << "x is: " << x;
    --x;
}
```

At the end of the loop body, control transfers to the top of the loop.

The loop body of a `do-while` is always executed at least once.

```
std::string str;
do {
    std::getline(std::cin, str);
} while(str != "breadfish");
```

Condition is executed at the **end** of the loop. If true, the loop body is executed again, if false, the loop ends.

for loop

Syntax:

```
for(initialization; condition; increment) body
```

- Initialization
  - Expressions or declarations
  - Executed **once** at the start of the loop
  - `int i = 0`
- Condition
  - Executed before each iteration of the loop
  - Like the condition of a `while` loop
- Increment
  - Executed after each iteration of the loop
  - Typically used to increment or decrement

```
for(int i = 0; i < 20; i++) {
    std::cout << i;
}
```

Note: the semicolons in for loop syntax are used in the "English" way, not the C++ way.

for loop

The three parts of the `for` loop are optional

```
for( ; i < 10; i++) { /* */ }
```
Loop with no initialization

```
for(int j = 0; j < 10; ) { /* */ }
```
Loop with no increment

```
for( ; ; ) { /* */ }
```
Loop with no condition (infinite loop)

Commas may be used to make multiple initializers or increments

```
for(int i = 0, j = 0; i < 10; i++, j++) { /* */ }
```

for loop

```
std::string str = "Marvelous Breadfish";
for(char c : str) {
    std::cout << c;
}
```

- Range-based `for`:
  - `for(`declaration `:` range`)` body
- Also known as “for-each” loop
- Used with collections, sequences, arrays

`return` statement

- Used within functions
- Transfer of control to calling code
- Two forms
  - Return value
  - Return void
- Covered more in function section

```
int main() {

    /* ... */
    return 0;
}

void sub() {

    /* ... */
    return;
}
```

`break` and `continue`

- Used within loops
- `break` used to end a loop prematurely
- `continue` used to skip (the rest of) a loop iteration

```
for(int x = 0; i < 10; x++) {
    if(x == 5)
        continue;
    std::cout << x;
    if(x == 8)
        break;
}
```

Note: use break and continue sparingly; there are generally better ways to achieve this behavior.

How can this code be rewritten without `break` and `continue`?

Functions

A parse farce of vexing lexing

What are functions?

- Groups of code that can be called to perform tasks
- `int main()` is a function
- Functions move common code to separate areas
- Functions may take input and carry out a procedure or produce a result
- Examples
  - `bool IsEven(int i)`
  - `std::string GetMessage()`

```
bool IsEven(int i) {
    return i % 2 == 0;
}

int main() {
    for(int i = 0; i < 10; i++) {
        std::cout << IsEven(i);
    }
}
```

Declaring and using functions

- Syntax
  - returnType name(param1, param2, ...) { body }
- Like variables, functions must be declared before they are used
  - This is unlike Java and other OO languages
- Functions may take zero, one, or more parameters
  - However, the parentheses are required in any case
- Call functions by supplying values for function parameters

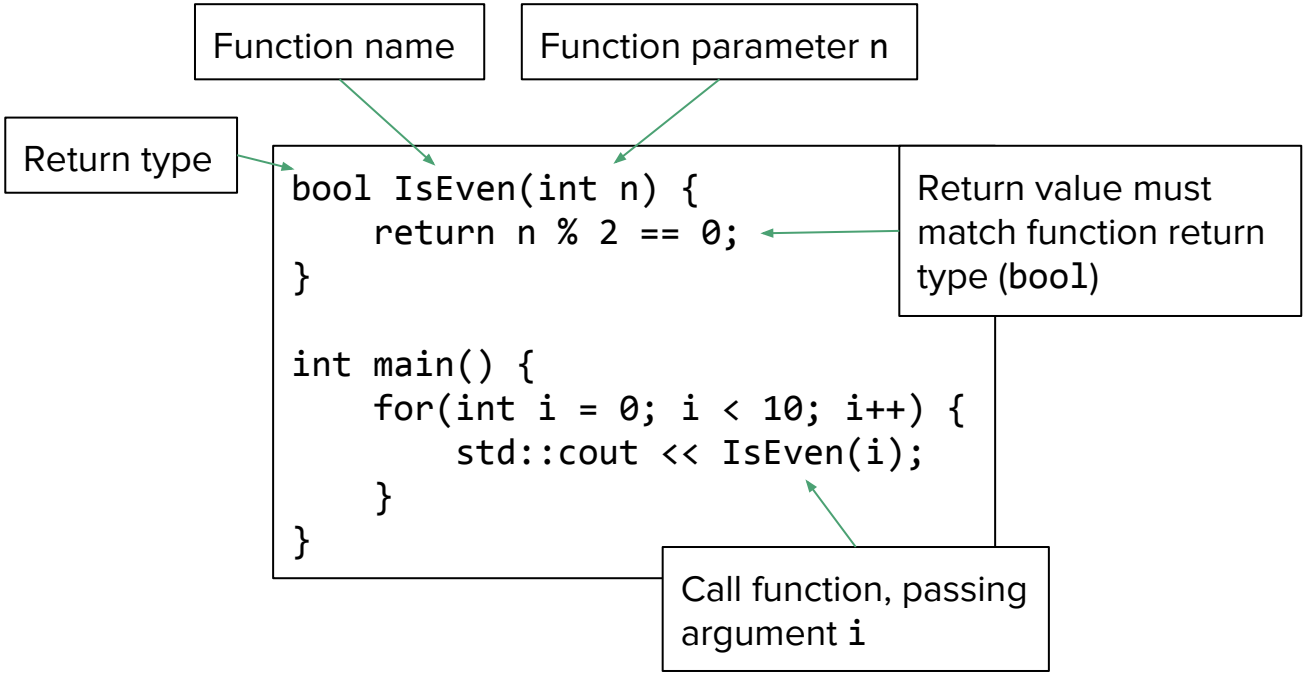
```
int Speed();

char CharAt(int index);

float Max(float a, float b);
```

```
int s = Speed();

char c = CharAt(s);

float f = Max(3.4f, 2.0f);
```

Forward declarations

- Functions may be declared many times before they are defined
- Used extensively in header (`.h`) files
- Also known as function prototypes

```
//function prototype
bool IsEven(int i);

//function definition
bool IsEven(int i) { return i % 2 == 0; }
```


Function name
Function parameter n
Return type
bool IsEven(int n) {
return n % 2 == 0;
}
int main() {
for(int i = 0; i < 10; i++) {
std::cout << IsEven(i);
}
}
Return value must match function return type (bool)
Call function, passing argument i

The value of a function

- Functions can return a value
  - Use `return` with expression
  - Expression must be of the same type as return type
  - `return` statement is required
- Functions can return no value
  - Use `return` without expression
  - Return type is `void`
  - `return` statement is optional

```
int ReturnInt() {
    std::cout << "returning 5";
    return 5;
}

void ReturnVoid() {
    std::cout << "no return";
}
```