Flow of control

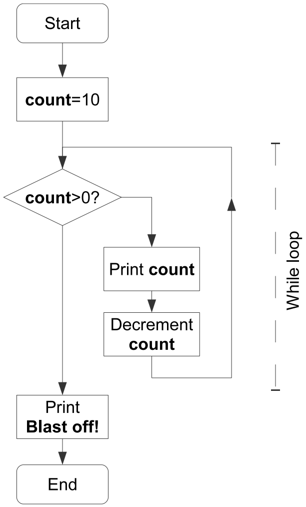
A while loop

```
void launchRocket() {
    int count = 10;
    while (count>0) {
        cout << count;
        cout << "\n";
        count--;
    }
    cout << "Blast off!\n";
}
```

The general syntax is:

```
while (<expression>) {
    <statements>
}
```

Flow chart


Start
count=10
count>0?
Print count
Decrement count
While loop
Print Blast off!
End

Another while loop

```
void printPowersOf2() {
    int count = 0;
    int currentPower = 1;
    while (currentPower<1000) {
        cout << "2^" << count << "=";
        cout << currentPower;
        cout << "\n";
        currentPower *=2;
        count++;
    }
}
```

Looping forever

```
void loopForever() {
    while (true) {
        cout << "Still looping\n";
    }
}
```

You will need to type **CTRL + C** to stop this running.

Do-while loops

```
void launchRocket_DoWhileVersion() {
    int count = 10;
    do {
       cout << count;
       cout << "\n";
       count--;
    } while (count>=1);
    cout << "Blast off!\n";
}
```

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

Do-while loops

The general syntax is:

```
do {
    <statements>;
} while (<expression>);
```

Anything you can do with a do-while loop could be done with a while loop anyway.

For loops

```
<initialize loop variables>;
while (<test loop variables>) {
    <perform main steps of code>
    <update loop variables>
}
```

This is captured in a for loop

```
for (<initialize loop variables>;
 <test loop variables>;
 <update loop variables>) {
    <perform main steps of code>
}
```

Example for loop

```
for (int i=10; i>0; i--) {
    cout << i;
    cout << "\n";
}
cout << "Blast off!\n";
```

Learn this by heart

```
for (int i=0; i<10; i++) {
    cout << i;
    cout << "\n";
}
```

In C++ you should:

- Start counting at 0.
- Use ++ to mean increment.
- Use a less than to decide when to stop.

Another for loop

Here's a for loop in steps of 10.

```
for (int i=0; i<100; i+=10 ) {
    cout << i;
    cout << "\n";
}
```

Which loop to use?

- Use `for` for simple loops with fixed end points and step size.
- Use `while` for complex and infinite loops.
- Use `do-while` only on the very rare occasions that it makes code easier to understand.

break

```
cout << "Enter positive numbers followed ";
cout << "by a negative number to quit\n";
int total = 0;
while (true) {
    int next;
    cin >> next;
    if (next<0) {
        break;
    }
    total += next;
}
cout << "The total is "<<total<<"\n";
```

continue

```
cout << "Enter positive numbers ";
cout << "Type CTRL+C to quit\n";
int total = 0;
while (true) {
    int next;
    cin >> next;
    if (next<0) {
        continue;
    }
    total += next;
    cout << "Positive total is "<<total<<"\n";
}
```

return

```
void countdown() {
    int i=10;
    while (true) {
        if (i==0) {
            return;
        }
        cout << i << "\n";
        i--;
    }
}
```

Using break, continue and return

Tip: Avoid `break` and `continue`

Most code is easier to read if you avoid using break, continue and early return statements.

Indicating an error

At the top of the file you should write

```
#include <stdexcept>
```

When an error has occurred write

```
throw logic_error("You can’t do that");
```

Provide some helpful text instead of “You can’t do that”.

Error handling example

```
double debitAccount( double balance, double amount ) {
    double newAmount = balance-amount;
    if (newAmount<0.0) {
        throw logic_error("No overdraft agreed");
    }
    return newAmount;
}
```

Question

Which is better

(A) a program that stops immediately when an error occurs;

(B) a program that attempts to continue when an error has occurred?

Advice on error handling

- Throw an error the moment you spot one.
- Nobody reads the log file.
- We're writing a maths library. Trying again doesn't make sense.
- Continuing in the face of errors is an advanced topic.

Switch statements

```
void printMessage(int score ) {
    switch (score){
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
            cout << "You have failed.\n";
            break;
        case 5:
        case 6:
            cout << "You have passed.\n";
            break;
        case 7:
            cout<< "Merit.\n";
            break;
        case 8:
        case 9:
            cout<<"Distinction.\n";
            break;
        default:
            cout<< "Invalid score.\n";
            break;
    }
    cout << "Good luck in your future career.\n";
}
```

Switch statements

- Avoid switch statements. They're hard to read.
- Avoid switch statements. They're a sign of code that will be hard to maintain. Use object orientation instead.
- Always add `break` statements and a `default` case.

★ What style rule did we break in the last slide?

Flow of control and &&

```
bool test1() {
    cout << "In test1\n";
    return false;
}

bool test2() {
    cout << "In test2\n";
    return true;
}

int main() {
    bool value = test1() && test2();
    cout << "Value "<<value<<"\n";
    return 0;
}
```

The ternary operator

```
int max( int a, int b ) {
    return a>b ? a : b;
}
```

Is equivalent to

```
int max( int a, int b ) {
    if (a>b) {
        return a;
    } else {
        return b;
    }
}
```

The ternary operator

```
<test expression> ? <value if true> : <value if false>
```

Avoid using the ternary operator.
If statements are easy to read. The ternary operator is hard to read.