Decisions and Loops

Flow of Control

Decisions

- Branching
  - Boolean logic
  - If statements
  - Switch statements
  - The evil *goto* statement
- Repetition
  - Conditional loops
    - while
    - do while
  - Counting loops
    - for loops

Branching

Branching

- Branching allows a program to make decisions based on some condition
  - If its raining, carry an umbrella
  - If height is greater than 6' do not permit entry
  - If x < 0 print an error message
- Conditions are written as *Boolean expressions*
  - That evaluate to *true* or *false*

If Statements

- An *if-else* statement chooses between two alternatives
  - Based on the value of a *condition*
  - If the condition is true the statement(s) following the *if* keyword are executed
  - Otherwise the statement(s) following the *else* keyword are executed
  - An *else* statement is not required

If Example

Let's assume that a program is controlling a missile, if the missile far away from the target it will attempt to evade any countermeasures, otherwise it will accelerate

```
if (range > 100){
      direction = evade();
}
else {
      speed = accelerate();
}
```

We are assuming the *range*, *direction* and *speed* variables have been declared, and that the *accelerate* and *evade* functions have been defined

If Statement Bodies

If the body of the if statement (the code that executes based on the condition) is only one line long then it does not need to be enclosed in **{}**s

```
if (range > 100)
        direction = evade();
else
        speed = accelerate();
```

This is OK but, be careful if you modify your code at a later date

If Statement Bodies

Consider another version of the previous example

This time, when the missile is close to the target it should accelerate and then explode, using a modified version of the previous example

```
if (range > 100)
        direction = evade();
else
        speed = accelerate();
        explode();
```

oops!

The indentation here is misleading, in fact the missile will explode *regardless of its range to the target*. As there are no {}s the bodies of the *if* and *else* statements only consist of the one line immediately following the condition

Boolean Logic

Boolean Expressions

- Conditions in if statements should be *Boolean expressions*
- Usually two operands compared using a comparison operator
  - One of ==, !=, <, >, <= , >=
  - Operators with two symbols (e.g. ==, <=) should not have spaces between the symbols
  - Make sure that you use == and not = as the test for equality

George Boole

- Born Nov. 2, 1815
  - Lincoln, England
- Died Dec. 8, 1864
  - Ballintemple, Ireland
- Philosopher and mathematician
- Boole approached logic by reducing it to algebra
- http://en.wikipedia.org/wiki/George_Boole

Comparison Operators

Operator	Meaning	Example	Result
>	greater than	`4 > 4`	False
<	less than	`'a' < 'p'`	True
>=	greater than or equal to	`4 >= 4`	True
<=	less than or equal to	`2.3 <= 2.2`	False
==	equal to	`4/3 == 1`	True
!=	not equal to	`4.0/3 != 1`	True

Boolean Variables

- There are two Boolean values
  - true and
  - false
- In C++, the *bool* type can be used to store Boolean values
  - `bool completed = false;`
  - Boolean values are often used to control program flow
  - cout prints bool values as 1 (true) or 0 (false)

Boolean Variables and Integers

- Integers are converted into bool values without generating compiler errors
  - Usually generating a compiler warning
    - This is dependent on compiler settings
  - Converting one type to another is referred to as *casting*
- Non-zero integer values are converted to true and zero to false
- True is converted to 1 and false to 0

Boolean Values in Conditions

- A Boolean expression is an expression that evaluates to true or false
- Therefore a condition in an if statement may consist of a single Boolean value
  - Or a function call that returns a bool variable
  - e.g. `if(completed) { …}`
    - Where `completed` is a Boolean value, that is presumably used to indicate is some process is finished

Logical Operators

- Multiple comparisons can be combined
  - Using *logical operators*
- These operators allow us to express decisions with more than one condition
  - AND, **&&** in C++
    - True only if *both* comparisons are true
  - OR, **||** in C++
    - True if *either* comparison is true
  - NOT, **!** in C++, negates a Boolean expression

True Or False?

- `12 < 13`
  - true
- `21 == 17`
  - false
- `23 != 21`
  - true
- `9 >= 8`
  - true
- `21 = 17`
  - true

- `12 < 13 && 7 > 7`
  - false
- `12 < 13 || 7 > 7`
  - true
- `!21 > 13 && 7 >= 7`
  - false
    - `!` has precedence over `>`
- `true || true && false`
  - true
    - `&&` has precedence over `||`

Truth Tables

- Boolean expressions can be evaluated in *truth tables*
- The symbol $\wedge$ represents *and*
- The symbol $\vee$ represents *or*
- The symbol $\neg$ represents *not* (has precedence over $\wedge$ and $\vee$)

e.g. !(12 < 13) && 7 > 7

p q

p	q	p∧q	p∨q	¬ p	¬ p∧q	¬ p∨q
T	T	T	T	F	F	T
T	F	F	T	F	F	F
F	T	F	T	T	T	T
F	F	F	F	T	F	T

Precedence Rules (abbreviated)

Operator (from high to low)	Associativity
[], (function), post++, post--, ., ->	left to right
++pre, --pre, !, * (dereference), &, unary - +	right to left
*, /, %	left to right
+, -	left to right
<<, >>	left to right
<, >, <=, >=	left to right
==, !=	left to right
&&	left to right
||	left to right
? : (conditional operator)	right to left
=, +=, -=, *=, /=, %=	right to left
, (comma operator)	left to right

Precedence Notes

- Brackets can be used to give operations precedence
- Binary operators with the same precedence are mostly evaluated left to right
  - Unary operators and *assignments* with the same precedence are evaluated right to left
- Note that x++ is *evaluated* before many other operations
  - But its primary (increment) *effects* occur later

Precedence Example

If x = 9, y = 9, and z = 3 what does this print?

true

true

false

```
if (++x > y || x > z && x > 13){
      cout << "Hippopotamus!";
}
else {
      cout << "Aardvark!";
}
```

false

Prints *Hippopotamus!* because the ++ has precedence over everything (making *x* equal to 10) and && has precedence over ||

If In Doubt …

- There are two very simple ways to avoid mistakes relating to precedence
  - Use brackets to avoid ambiguity
  - Change complex statements into multiple simpler statements
- And one more piece of advice, be careful about type conversions

Pitfall: x < y < z

This logical (not C++) statement: 0 < x < 100 is true if x is between 0 and 100. There is no shorthand equivalent in C++, unfortunately such an expression ***is*** legal …

```
// Set adult to true age is 19 to 60
int age = 12;
bool adult;
if (18 < age < 61){
      adult = true;
}
else {
      adult = false;
}
```

oops!

Except that it doesn't work correctly!

This always prints true!

Think of the condition as ((18 < age) < 61)

If age is 12 then (18 < age) is false, so now evaluate: false < 61

false is cast to the an *int* value of 0, and when we evaluate 0 < 61 we get true

Short Circuit Evaluation

- Where two expressions are joined by && or || C++ performs *short circuit evaluation*
  - It may be possible to determine the result of the entire expression from the first expression
  - If so, the second expression is not evaluated
    - e.g. (age > 19 && age < 61)
    - If age is 17 there is no point in checking to see if age is < 61 as the entire expression must be false

Pitfall: Assignment not Equals

One of the philosophies underlying C++ is that it will not perform checks to see if what you are doing is sensible (because such checks reduce efficiency)

```
int password;
cout << "Enter your password Mr. Trump: ");
cin >> password;
if(password = 1121865){
        cout << "You may now commence nuclear war" << endl;
}else{
        cout << "INTRUDER ALERT!";
}
```

should be ==

output:

```
Enter your password Mr. Trump: 91
You may now commence nuclear war
```

Here password is *assigned* the value of **1121865**, and then the condition is evaluated

It uses the value of password (now **1121865**), which evaluates to true!

Multiple Branches

Multiple Options

- An if-else statement chooses between just two alternatives
  - It is often useful to allow more than two alternatives
  - This can be achieved in a number of ways
    - Multiple if-else statements
    - Nested if-else statements
    - If - else if – else statement
    - Switch statements

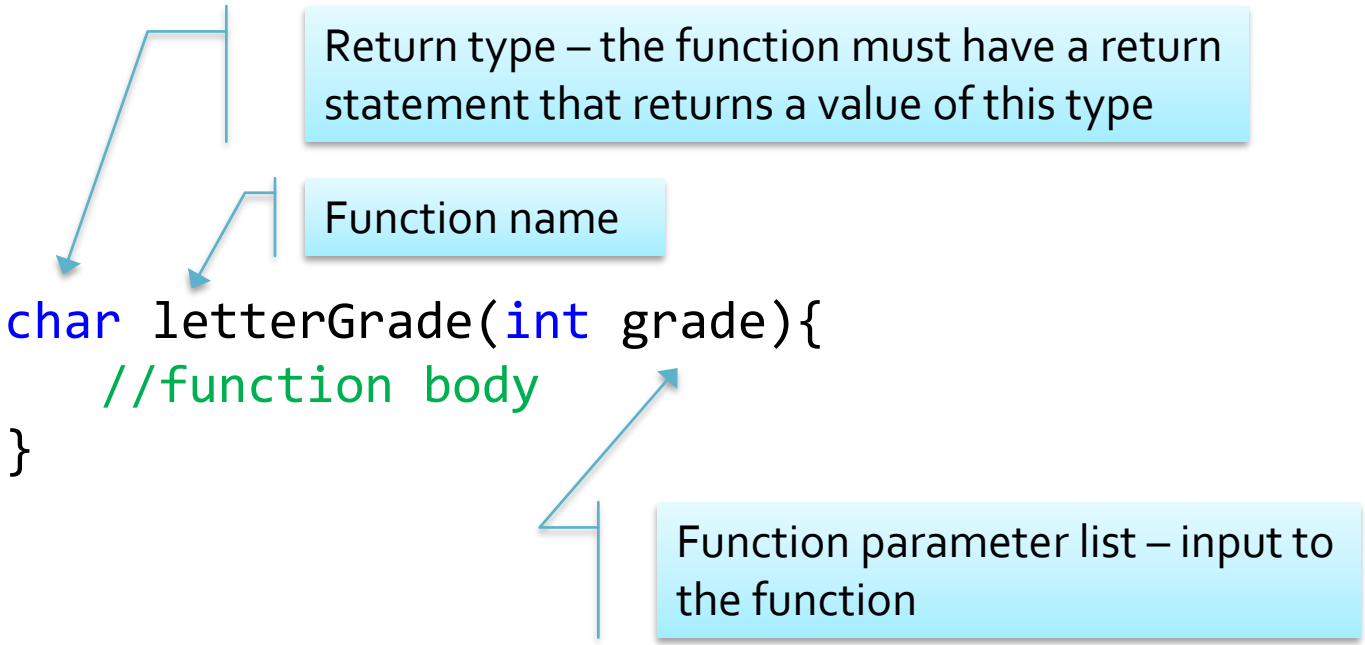
Assigning Grades

- Write a function to *print* the letter grade of a student given the numeric grade
  - Following this grade scheme*
    - 0 – 49 fail
    - 50 – 59 D
    - 60 – 74 C
    - 75 – 89 B
    - 90+ A
- *Not the grade scheme used in this class

Quick Function Introduction

- We could have just written code to print letter grades from numeric grades
  - Instead we will write a function that returns a letter grade
  - For a quick introduction to functions
    - Which we will see a lot more of later
- A function is a separate block of code
  - That can be used in the main function
    - Or other functions

Function Anatomy



The function might be called (used) like this to print a letter grade:

```
cout << letterGrade(87);
```

Multiple If Else (1)

```
// Function to return a letter grade
char letterGrade(int grade){
    if (grade > 50)
        return 'D';
    if (grade > 59)
        return 'C';
    if (grade > 74)
        return 'B';
    if (grade > 89)
        return 'A';
    else //(grade < 50)
        return 'F';
}
```

The function contains 4 separate if statements, one with an else clause

It usually returns the wrong grade

A function terminates *as soon as* a return statement is reached

Consider a grade of 81

81 is over 50 the first if statement's condition is true so *D* is returrned

Multiple If Else (2)

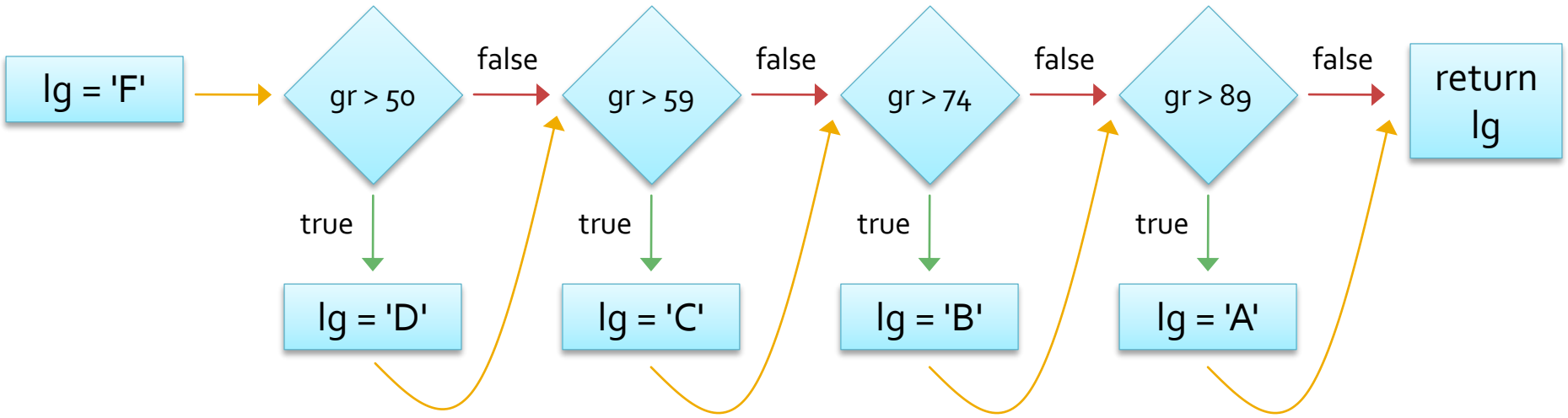
```
// Function to return a letter grade
char letterGrade(int grade){
    char letterGrade = 'F';
    if (grade > 50)
        letterGrade = 'D';
    if (grade > 59)
        letterGrade = 'C';
    if (grade > 74)
        letterGrade = 'B';
    if (grade > 89)
        letterGrade = 'A';
    return letterGrade;
}
```

The function contains 4 separate if statements, none with an else clause

It returns the correct grade

It makes 4 comparisons before returning the letter grade

Multiple If Flowchart




```
char letterGrade(int grade){
   char letterGrade = 'F';
      if (grade > 50)
            letterGrade = 'D';
      if (grade > 59)
            letterGrade = 'C';
      if (grade > 74)
            letterGrade = 'B';
      if (grade > 89)
            letterGrade = 'A';
      return letterGrade;
}
```

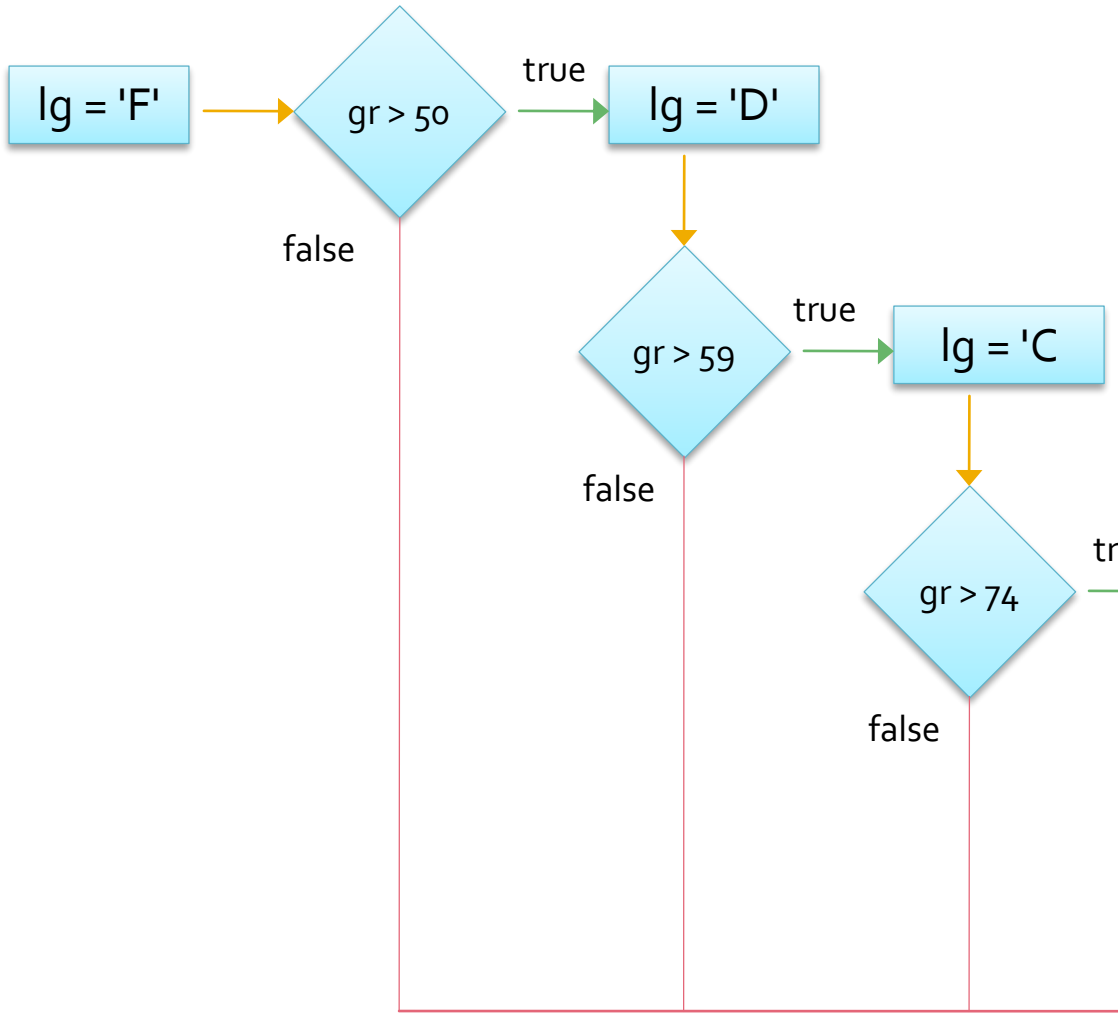
Nested If Else

```
// Function to return a letter grade
char letterGrade(int grade){
    char letterGrade = 'F';
    if (grade > 50){
        letterGrade = 'D';
        if (grade > 59){
            letterGrade = 'C';
            if (grade > 74){
                letterGrade = 'B';
                if (grade > 89){
                    letterGrade = 'A';
    }   }   }   }
    return letterGrade;
}
```

This version is similar to the previous one except that the if statements are *nested* within each other

If grade is 45 only one comparison would be made

Nested If Flowchart




```
char letterGrade(int grade){
   char letterGrade = 'F';
   if (grade > 50){
      letterGrade = 'D';
      if (grade > 59){
         letterGrade = 'C';
            if (grade > 74){
               letterGrade = 'B';
               if (grade > 89){
                  letterGrade = 'A';
   }   }   }   }
   return letterGrade;
}
```

If Else If Else (1)

```
// Function to return a letter grade
char letterGrade(int grade){
    char letterGrade;
    if (grade > 50)
        letterGrade = 'D';
    else if (grade > 59)
        letterGrade = 'C';
    else if (grade > 74)
        letterGrade = 'B';
    else if (grade > 89)
        letterGrade = 'A';
    else // (grade < 50)
        letterGrade = 'F';
    return letterGrade;
}
```

In an *if – else if – else* statement only the body of the *first* true condition is evaluated

So this only ever returns D or F!

Let's do what we could have done a while ago, *reorder the statements*

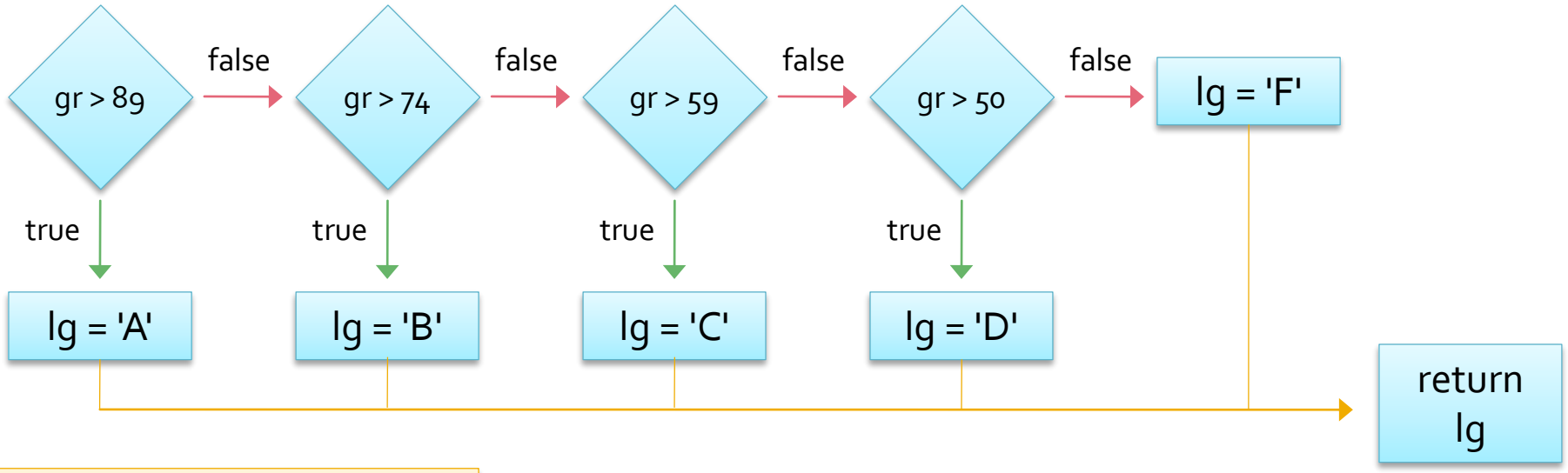
If Else If Else (2)

```
// Function to return a letter grade
char letterGrade(int grade){
    char letterGrade;
    if (grade > 89)
        letterGrade = 'A';
    else if (grade > 74)
        letterGrade = 'B';
    else if (grade > 59)
        letterGrade = 'C';
    else if (grade > 50)
        letterGrade = 'D';
    else // (grade < 50)
        letterGrade = 'F';
    return letterGrade;
}
```

This works correctly, the same plan could have been used to fix the first version

The function checks for an A, then for a B, then for a C, then for a D, if none of those apply the grade must be an F (*else*)

If Else Flowchart




```
char letterGrade(int grade){
   char letterGrade;
     if (grade > 89)
           letterGrade = 'A';
     else if (grade > 74)
           letterGrade = 'B';
     else if (grade > 59)
           letterGrade = 'C';
     else if (grade > 50)
           letterGrade = 'D';
     else // (grade < 50)
           letterGrade = 'F';
     return letterGrade;
}
```

Matching Else

- An if statement does not have to have an associated else statement
  - An else statement does need an associated if statement
- Else statements are always matched to the closest if (or else if) statement
  - Regardless of indentation
  - Control with {}s if necessary

Conditional Operator ?

- C++ has a shorthand for an if else statement
  - The conditional operator
- The operator is a two part operator with three operands
  - A *ternary* operator
- The operator consists of the following
  - A Boolean expression followed by a ?
  - A value
  - A colon (:) followed by a value

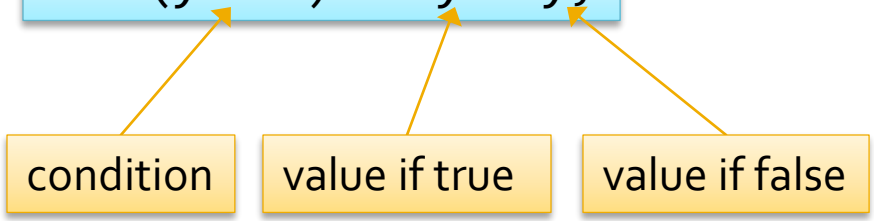
? Example

store the absolute value of x in y

```
if (y < 0)
        x = -y;
else
        x = y;
```

the equivalent conditional expression

```
x = (y < 0) ? -y : y;
```



Switch Statements

Switch Statements

- Switch statements can be used to choose between different cases
  - As long as the cases can be evaluated to an integer or a character
- As another, grade-related, example let's write a function to print a message
  - That varies based on the grade

Switch Example (1)

```
// Switch statement to print a letter grade message
char letterGrade = 'B';
switch (letterGrade)
{
case 'A':
        cout << "Wow, an A, congratulations!" << endl;
case 'B':
        cout << "Well done, you got a B" << endl;
case 'C':
        cout << "You passed, with a C" << endl;
case 'D':
        cout << "It was close but you passed, a D" << endl;
case 'F':
        cout << "Too bad, so sad, you failed" << endl;
default:
        cout << "Error!\n" << endl;
}
```

The statement after the first matching case is executed, then *all of the following statements*

output

```
Well done, you got a B
You passed, with a C
It was close but you passed, a D
Too bad, so sad, you failed
Error!
```

Switch Example – Fixed

```
// Switch statement to print a letter grade message
char letterGrade = 'B';
switch (letterGrade)
{
case 'A':
          cout << "Wow, an A, congratulations!" << endl;
          break;
case 'B':
          cout << "Well done, you got a B" << endl;
          break;
case 'C':
          cout << "You passed, with a C" << endl;
          break;
case 'D':
          cout << "It was close but you passed, a D" << endl;
          break;
case 'F':
          cout << "Too bad, so sad, you failed" << endl;
          break;
default:
          cout << "Error!\n" << endl;
}
```

The *break* statement immediately moves to the end of the switch statement body (the closing **}**)

But, this won't work if we set letterGrade to 'b'

Switch Example – Multiple Cases

```
// Switch statement to print a letter grade message
char letterGrade = ‘b';
switch (letterGrade)
{
case 'A': case 'a':
          cout << "Wow, an A, congratulations!" << endl;
          break;
case 'B': case 'b':
          cout << "Well done, you got a B" << endl;
          break;
case 'C': case 'c':
          cout << "You passed, with a C" << endl;
          break;
case 'D': case 'd':
          cout << "It was close but you passed, a D" << endl;
          break;
case 'F': case 'f':
          cout << "Too bad, so sad, you failed" << endl;
          break;
default:
          cout << "Error!\n" << endl;
}
```

Switch Limitations

- The *switch* test expression and the *case* labels must be integer values
  - Which includes the *char* type
- Therefore *switch* statements cannot be used in the following situations
  - To evaluate floats, or other non integer types
  - To evaluate ranges of values
    - At least without creating many cases

The goto Statement

The Goto Statement

- C++ has a *goto* statement
  - That directs execution to a labelled statement in the same function
- Avoid the use of goto statements
  - They are unnecessary and their effect can be achieved by the use of other constructs
  - Goto statements make programs very difficult to read, and therefore hard to modify

Repetition

Loop Introduction

- Loops allow statements to be repeated
  - The code to be repeated is in the loop body
  - One repetition of the body is called an *iteration*
- Loops are structurally similar to if statements
  - The loop control statement(s) are contained in ( )s after the keyword
  - The loop body is contained in { }s

While Loop

- A *while* loop consists of the keyword **`while`**, a condition and a loop body
  - The condition is a Boolean expression
    - Just like an if statement condition
  - The loop iterates until the condition is no longer true (*while* it is true)
- The loop body should include code that eventually makes the condition false
  - Or the loop will iterate for ever (an *infinite loop*)

Factorials

- Computing a factorial is a repetitive process
  - 5! = 1*2*3*4*5 = 120
    - Set *result* = 1, *i* =2
    - Multiply result by *i*
    - Add one to *i*
    - Repeat until *i* > 5
- We will go through some alternative versions of a function to compute factorials
  - Writing a function creates a self contained construct that could be used in a program

While Factorial

Write a function that returns the factorial of the integer parameter:
e.g. fact(5) = 5! = 5 * 4* 3 * 2 * 1 = 120

```
// PRE: x must be a +ve integer
// Function that returns the factorial of x
long long fact(int x){
        long long result = 1;
        int i = 2; //loop control variable
        while (i <= x){
                result = result * i;
                ++i;
        }
        return result ;
}
```

a large integer

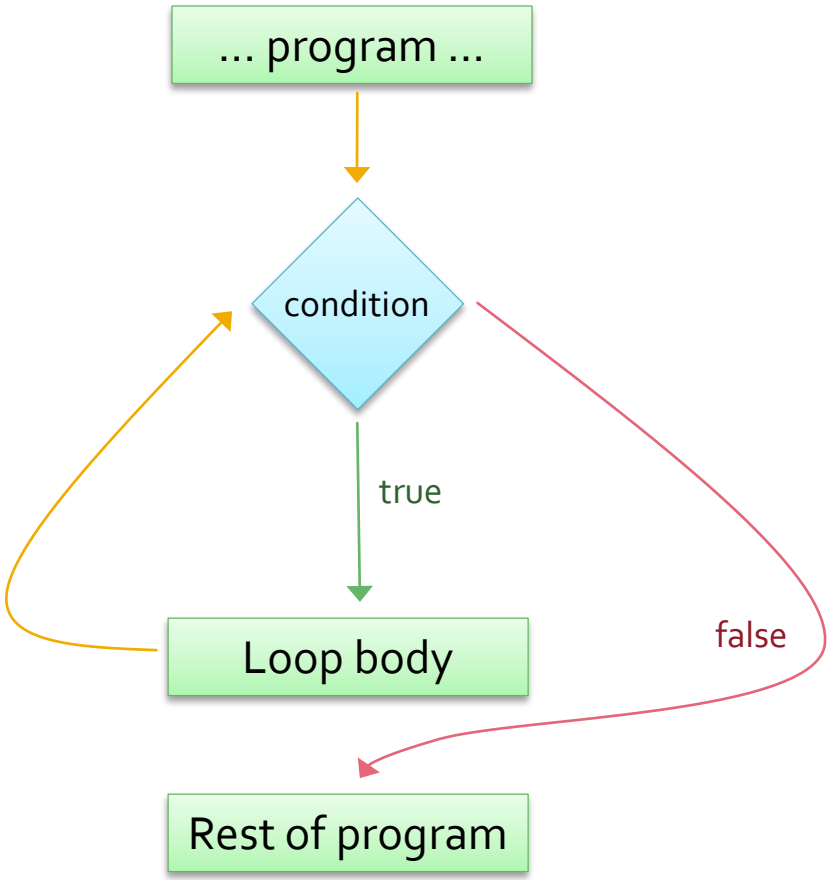
don't put a ; here – it makes an empty loop

i must be incremented in the loop

The function does not handle –ve numbers

Note the indentation of the statements in the function and the loop

While Flowchart



- `while` statements contain a condition
  - If the condition is true the body is executed
  - Then the condition is tested again
  - If the condition is false the program continues from the end of the loop body

Entry Condition

- A while loop has an *entry condition*
  - If the condition is initially false the loop body will not be processed at all
- Sometimes the first iteration should occur outside of and before the loop
  - So that the variable being evaluated in the condition can be initialized appropriately

Summing Numbers

Write a function that sums numbers until the user enters o

```
// Function that returns the sum of values
// entered by the user
int sum(){
        int result = 0;
        int next;
        cout << "Enter a number, 0 to end");
        cin >> next;
        while (next != 0){
                result += next;
                cout << "Enter a number, 0 to end";
                 cin >> next;
        }
        return result ;
}
```

Because *next* controls the loop it needs a value before the loop starts

Although there are some alternatives:

Initialize next to non-zero and then get input before adding to result, or

Use a do … while loop

Do While Loop

- A *do … while* loop's condition comes *after* the loop body
  - The loop body will iterate at least once
  - Whereas a while loop will not iterate at all if the condition is initially false
- Any *do … while* loop can be replaced by *while*
  - Possibly needing some extra statements before the loop statement
  - Some people prefer *while* loops because the condition comes first

Do While Factorial

Write a function that returns the factorial of the integer parameter:
e.g. fact(5) = 5! = 5 * 4* 3 * 2 * 1 = 120

```
// PRE: x must be a +ve integer
// Function that returns the factorial of x
long long fact2(int x){
        long long result = 1;
        int i = 1; //loop control variable
        do {
                result = result * i;
                ++i;
        } while (i <= x);
        return result ;
}
```

note the ; after the condition

Counting Loops

- Loop bodies are often repeated a certain number of times
  - Rather than ending at an indeterminate time
    - e.g. factorial function, processing the values in a list
- *For* loops can be used to iterate a given number of times
  - By incrementing an integer variable
  - And ending when the variable reaches a value
  - For loops can do anything that while loops can

For Factorial

Write a function that returns the factorial of the integer parameter:
e.g. fact(5) = 5! = 5 * 4* 3 * 2 * 1 = 120

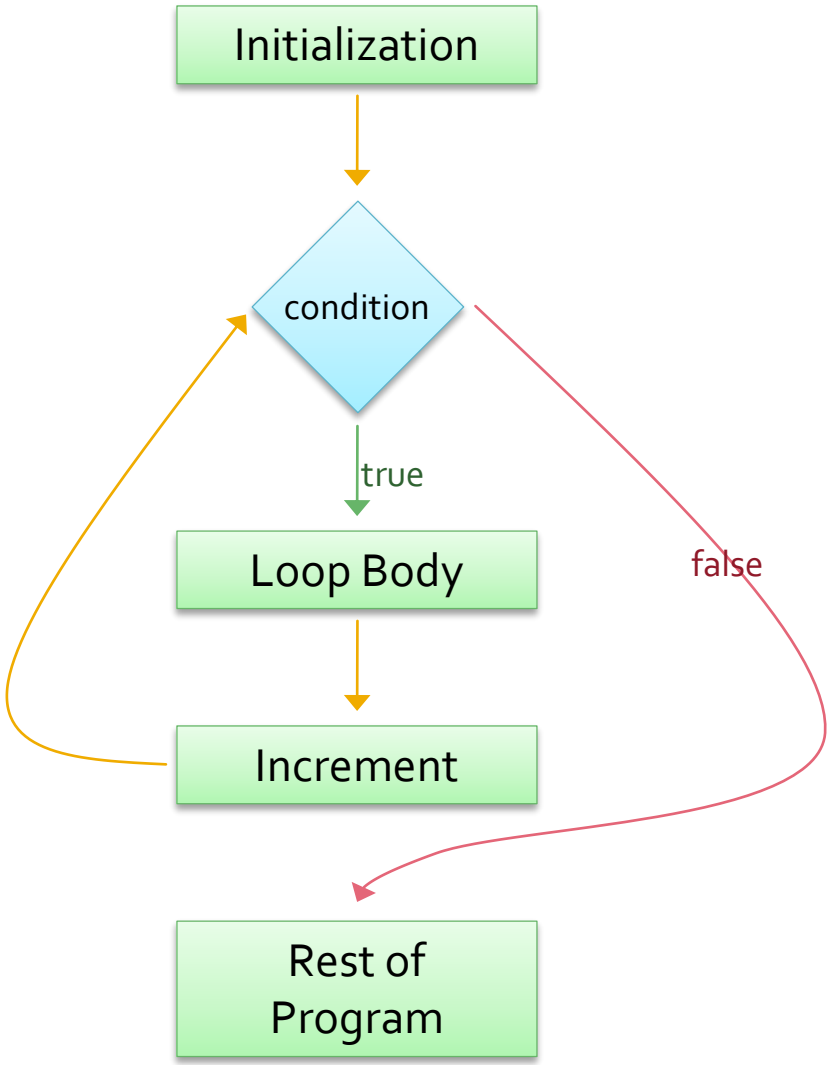
```
// PRE: x must be a +ve integer
// Function that returns the factorial of x
long long fact3(int x){
        long long result = 1;
        for (int i = 2; i <= x; ++i){
                result = result * i;
        }
        return result ;
}
```

The loop control statement consists of three statements

initialization | condition | increment

In this example the loop control variable is also declared in the initialization statement

Controlling For Loops



- `for` statements consist of three expressions
  - Separated by `;`s
- Initialization
  - Executed only once
- Condition
  - Tested before each iteration
    - The last time the condition is tested there is no iteration
    - Since the test returns false
- Increment
  - Applied after each iteration

More About for Loops

- It is usual to use *for* loops as *counting loops*
  - Initialize the loop control variable
  - Test to see if the end of the count is reached
  - Increment the count (the loop control variable)
- The *for* loop structure is much more general
  - An expression evaluated once at the beginning
  - A condition that is evaluated *before* each iteration
    - The body is only executed if the condition is true
  - An expression evaluated once *after* each iteration

Comma Operator

- The comma operator evaluates a list of expressions returning the last expression
  - e.g. $z = (x = 1, y = x + 1);$
  - $x = 1$ is evaluated first (assigning 1 to x)
  - $y = x + 1$ is then evaluated (assigning 2 to y)
  - The comma expression returns the value of y
    - So z is also assigned 2
- It is most commonly used in for loops
  - To allow multiple initialization or increment statements

break and continue

- The `break` and `continue` statements can be used to change flow of control
- The break statement terminates the processing of a loop or switch statement
  - It ends evaluation of its enclosing body
  - And switches control to the next statement after the closing `}`
- The continue statement terminates the processing of the current loop iteration
  - And then continues with the loop, first testing its condition
  - Like *goto*, *continue* and *break* can make programs harder to understand

Pre and Post Increment

- There are two version of the increment operator
- Pre: ++x
  - The variable is incremented and then the rest of the statement is executed
- Post: x++
  - The variable is incremented only after the rest of the statement has been executed
- If the statement consists only of the increment operator there is no difference between the two
- All of this applies to the decrement operator

Nested Loops

- Like if statements, loops can be nested
  - One loop can contain another loop
- The use of functions can make nested loops easier to understand
  - Particularly if one of the nested loops is placed inside a function
- Nested loops may cause a program to run slowly
  - But may also be unavoidable

Which Loop?

- There is a large element of choice
  - What you can do with one loop, you can do with another
- *For* loops are a natural choice for counting with an index
- *While* loops are a natural choice for *indefinite iteration*
  - Such as when the loop ends based on user input

More About Loops

- There is another version of the for loop
  - That iterates over the values in a container
  - This will be introduced when we discuss arrays
- We will also cover more complex examples of loops
  - Including some nested loops
  - Again, when we discuss arrays