Outline

- Conditional statements (if)
- More on loops
- break and continue
- Functions
- Reading and writing from files
Conditional statements

We can use the `if` statement to execute code only if the result of an expression is `true`

```java
if (x < 5)
{
    // code here executed only if x < 5
}
```

- The expression usually involves a comparison operator `<`, `>`, `<=
or `>=`, `==` or `!=`
- The logical AND (`&&`), logical OR (`||`) and logical NOT (`!`) operators can be used to combine conditions, e.g.

```java
if (!(x>=0 && x<n))
{
    // code here executed if x is NOT
    // between 0 and n-1, inclusive
}
```
The if statement

The optional `else` statement specifies code that is executed if the condition is `false`

```cpp
if (x < 5) {/*code here executed only if x < 5*/}
else
{
    // code here executed only if x >= 5
}
```

The instruction after an `else` statement can be another `if`:

```cpp
if (x < 5) {/*executed only if x < 5*/}
else if (x < 10) {/*executed only if 5 <= x < 10*/}
else if (x < 20) {/*executed only if 10 <= x < 20*/}
else {/*executed only if x >= 20 */}
```

See also the `switch` statement (C++ Primer §6.6)
Recall the **while** loop from last time:

```c
int i = 1;
while (i<=nTerms) {
    sum += 1.0/(i*i);
    i++;
}
```

- We set up a counter `int i=1;`
- If `i<=nTerms` we execute the loop body (in `{ }`)  
- At the end of the loop body we increase `i` by 1, and check whether to run the loop again.
For loops

Instead of a `while` loop,

```c
int i = 1;
while (i<=nTerms)
{
    sum += 1.0/(i*i);
    i++;
}
```

we could use an equivalent `for` loop,

```c
for (int i=1; i<=nTerms; i++)
{
    sum += 1.0/(i*i);
}
```

for ([initial]; [condition]; [iteration])

- A `while` statement is followed by a condition in parentheses
- A `for` loop is followed by three expressions, separated by semicolons ;
For loops

Instead of a \texttt{while} loop,

\begin{verbatim}
int i = 1;
while (i<=nTerms)
{
    sum += 1.0/(i*i);
i++;
}
\end{verbatim}

we could use an equivalent \texttt{for} loop,

\begin{verbatim}
for (int i=1; i<=nTerms; i++)
{
    sum += 1.0/(i*i);
}
\end{verbatim}

\texttt{for} \ ([initial]; [condition]; [iteration])

\begin{itemize}
  \item The [initial] statement, \texttt{int \ i=1;}, is executed before the loop begins
\end{itemize}
For loops

Instead of a `while` loop,

```c
int i = 1;
while (i<=nTerms)
{
    sum += 1.0/(i*i);
    i++;
}
```

we could use an equivalent `for` loop,

```c
for (int i=1; i<=nTerms; i++)
{
    sum += 1.0/(i*i);
}
```

- The `[condition]` statement, `i<=nTerms`, is tested before each loop, as in the `while` statement
For loops

Instead of a *while* loop,

```c
int i = 1;
while (i<=nTerms)
{
    sum += 1.0/(i*i);
    i++;
}
```

we could use an equivalent *for* loop,

```c
for (int i=1; i<=nTerms; i++)
{
    sum += 1.0/(i*i);
}
```

*for* ([initial]; [condition]; [iteration])

- The [iteration] statement, i++, is executed at the end of the loop body
For loops

Instead of a `while` loop,

```cpp
int i = 1;
while (i<=nTerms)
{
    sum += 1.0/(i*i);
    i++;  
}
```

we could use an equivalent `for` loop,

```cpp
for (int i=1; i<=nTerms; i++)
{
    sum += 1.0/(i*i);
}
```

See also: `do while` loops (C++ Primer, §6.9)
Break

The `break` statement is used to immediately exit a `for`, `while` or `do while` loop.

```c
for (int i=1; i<=nTerms; i++)
{
    double increment = 1.0/(i*i);
    sum += increment;
    // stop adding terms once they get sufficiently small
    if (increment < 1.0e-8)
    {
        break;
    }
}
```

- `break` is almost always executed conditionally using an `if` statement
- In nested loops, `break` only exits the innermost loop
Continue

The `continue` statement is used to immediately move to the next iteration of a `for`, `while` or `do while` loop.

```cpp
for (int i=1; i<=nTerms; i++)
{
    // Don't sum terms where i is a multiple of 3
    if (i%3 == 0)
    {
        continue;
    }
    sum += 1.0/(i*i);
}
```

- `continue` is almost always executed conditionally using an `if` statement
- In nested loops, `continue` only affects the innermost loop
- The loop condition is still tested in the loop following a `continue` statement
Functions

Functions are named sections of code can be called from elsewhere in the program.

Functions optionally
- take parameters
- evaluate to (or return) a value

```c++
double power(double v, int n) {
    double r = 1;
    for (int i=0; i<n; i++) r *= v;
    return r;
}

int main() {
    std::cout << power(5.0, 2) << std::endl;
    std::cout << power(3.0, 3) << std::endl;
    return 0;
}
```
Why use functions?

Functions allow *abstraction*:

- So long as we know what a function does, we don’t have to understand exactly how it works
- This is especially true if the function has no *side-effects*

Functions avoid repetition of frequently-used code

- Easier to modify and debug, if we only have to change code in one location

Functions improve the clarity of the code

- We can write ‘high-level’ code
Defining functions

```c
[return_type] [function name]( [parameters] )
{
   [function body]
   [return_statement]
}
```

```c
double power(double v, int n)
{
   double r = 1;
   for (int i=0; i<n; i++) r *= v;
   return r;
}
```

- The function name can include letters (either case), numbers and the underscore ( _ )
- ...but it cannot start with a number.
- (This is the same rule as for variable names)
Defining functions

```plaintext
[return_type] [function name]( [parameters] )
{
   [function body]
   [return_statement]
}
```

double power(double v, int n)
{
   double r = 1;
   for (int i=0; i<n; i++) r *= v;
   return r;
}

- Function arguments are variables, with a type and a name.
- Arguments are separated with a comma.
- The arguments are in scope only within the function body
- Functions with no arguments still require the parentheses:
  function()
Defining functions

```
[return_type] [function name]([parameters])
{
    [function body]
    [return_statement]
}
```

```c
double power(double v, int n)
{
    double r = 1;
    for (int i=0; i<n; i++) r *= v;
    return r;
}
```

The syntax here is for parameters *passed by value*

- The arguments can be changed within the function
- Doing so does not change their values outside the function
- We can also *pass by reference* and *pass by const-reference* (more on this later)
Defining functions

[code]
[return_type] [function name]([parameters])
{
    [function body]
    [return_statement]
}

double power(double v, int n)
{
    double r = 1;
    for (int i=0; i<n; i++) r *= v;
    return r;
}
[/code]

- The return type of a function is the type that the function evaluates to when called
- If the return type is specified as `void`, no value is returned
- Functions that do not return values are known as void functions or procedures
Defining functions

```
define [return_type] [function name]([parameters])
{
    [function body]
    [return_statement]
}
```

```c
double power(double v, int n)
{
    double r = 1;
    for (int i=0; i<n; i++) r *= v;
    return r;
}
```

- The function body can contain any code
- Functions can call other functions, including themselves (allowing recursion)
Defining functions

\[
\texttt{[return\_type]} \quad \texttt{[function\_name]}(\texttt{[parameters]})
\]
\{

[function\_body]

\texttt{[return\_statement]}  
\}

\begin{verbatim}
double power(double v, int n)
{
    double r = 1;
    for (int i=0; i<n; i++) r *= v;
    return r;
}
\end{verbatim}

A return statement

- specifies the value that the function evaluates to (this return value is of type \texttt{[return\_type]})
- exits the function and returns to the code that called it
Defining functions

```
[return_type] [function name]([parameters])
{
    [function body]
    [return_statement]
}
```

```c
double power(double v, int n)
{
    double r = 1;
    for (int i=0; i<n; i++) r *= v;
    return r;
}
```

- `return` statements can occur anywhere in the function body (usually within `if` statements)
- A void function needs no `return` statement
  - but used without an argument (`return ;`) to exit the function
Example functions

Two **double** arguments, no return value (**void**):

```cpp
// Function to display two doubles in coordinate form (x, y)
void DisplayCoordinate(double x, double y)
{
    std::cout << "(" << x << ", " << y << ")";
    // no return statement needed for void return type
}
```

Two **int** arguments, returns an **int**

```cpp
int Maximum(int a, int b) // returns max(a, b)
{
    if (a>b) return a;
    else return b;
}
```
Example functions

Two unsigned arguments, unsigned return value:

// Recursive function to return the greatest common divisor
// of its arguments. For non-negative integers, so uses unsigned
unsigned GCD(unsigned x, unsigned y)
{
    if (y == 0)
    {
        return x;
    }
    else return GCD(y, x%y);
}

No arguments, int return value:

// The 'main' function that is the entry point of a C++ program
int main()
{
    // code here
    return 0;
}
Function declarations

Functions must be *declared* in code before they are called

- Either the whole function can be put above the place it is called...

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

int main()
{
    std::cout << Maximum(3, 5) << std::endl; // function call
    return 0;
}
```
Function declarations

Functions must be *declared* in code before they are called

- ...or the *declaration* (or *prototype*) can be put above the place that the function is called, and the function *definition* below:

```cpp
// function declaration: no function body, and a semicolon
int Maximum(int a, int b);

int main()
{
    std::cout << Maximum(3, 5) << std::endl; // function call
    return 0;
}

// function definition: has function body, and no semicolon
int Maximum(int a, int b)
{
    if (a>b) return a;
    else return b;
}
```
Global variables

If a variable needs to be accessed by several functions, we can define it outside the functions as a *global variable*:

```cpp
int count = 0; // count is global, defined outside all functions

void IncrementCount()
{
    count++; // use count in the IncrementCount function
}

int main()
{
    IncrementCount();
    std::cout << count << std::endl; // and in main
    return 0;
}
```

It is usually preferable to use *classes* (discussed later) rather than global variables.
Streams

- Recall that `std::cout` displays text to the screen, using the operator `<<`:

```cpp
int a = 5;
std::cout << "The value of a is " << a << std::endl;
```

- This syntax is common to all output streams

- **Output streams** are used for **writing** formatted text to
  - standard output (`std::cout`), which displays on the screen
  - files on disk (using objects of type `std::ofstream`)
  - text strings in memory (using objects of type `std::ostringstream`)

- Also **input streams**, used for **reading** data from
  - standard input (`std::cin`), which reads from the keyboard
  - files on disk (using `std::ifstream` objects)
  - text strings (using `std::istringstream` objects)
Writing to files

There are three stages to saving data to disk:

• ‘Open’ a file for writing
  • this creates (or overwrites) a file on disk with a given file name
  • this step can fail if the file name is invalid, if the disk cannot be written to, or if another program has the file open
  • we should therefore check that the file is opened properly before proceeding further

• Write data to this file
  • Data is not necessarily written to disk immediately, since writing to disk is relatively slow. Instead, the data is stored in memory (buffered) and written to disk once there is enough of it

• ‘Close’ the file
  • This first flushes the file, writing all remaining data to disk
Writing to file streams

Output file streams (std::ofstream) are used write to a file:

```cpp
#include <fstream>  // std::ofstream is in the <fstream> header

int main()
{
    // declare a variable of type std::ofstream named 'file'
    std::ofstream demoFile;

    // try to open the file for writing, specifying the filename
    demoFile.open("outputDemonstration.txt");
    // if the file failed to open, return from main with nonzero
    if (!demoFile) return 1;

    // write some text to the file; syntax is like std::cout
    demoFile << "Text to be saved in the file.\n";

    demoFile.close(); // close the file (saves text to disk)

    return 0; // return from main with zero (success)
}
```
Formatting output streams

The way that `std::cout` and other output streams display (floating-point and integer) numbers can be customised:

- Columnated numerical data:

```cpp
std::cout.width(10); std::cout << 1;
std::cout.width(10); std::cout << 3.56;
std::cout.width(10); std::cout << 0.356735 << std::endl;

std::cout.width(10); std::cout << 1.234;
std::cout.width(10); std::cout << 3.56;
std::cout.width(10); std::cout << 0.356735 << std::endl;
```

Output:

```
1   3.56  0.356735
1.234 3.56  0.356735
```
Formatting output streams

The way that `std::cout` and other output streams display (floating-point and integer) numbers can be customised:

- **Number of significant figures:**

  ```cpp
double pi = 3.1419265358969323;
std::cout.precision(5);
std::cout << pi << std::endl;
std::cout.precision(15);
std::cout << pi << std::endl;
```

**Output:**

```
3.1416
3.1419265358969
```

For further details, see:

- `stream_formatting.cpp` on the course website
- C++ Primer §A3.3
Input streams

- We use input streams to read data into variables
- `std::cin` (the ‘standard input’, which reads from the keyboard) illustrates input stream syntax:

```cpp
#include <iostream> // for std::cin and std::cout

int main()
{
    int a;
    std::cout << "Enter a number: ";

    // read a number from the keyboard: note >> (not <<)
    std::cin >> a;

    std::cout << a << "^2" << "=" << (a*a) << std::endl;
    return 0;
}
```
Input streams

- We can read more than one variable at once:

```cpp
int a;
double b;
std::cin >> a >> b;
```

Variables are separated in the input by whitespace (spaces, tabs, and/or carriage returns)

- What if the input is not a valid number? We can check this with:

```cpp
int a; double b;
if (std::cin >> a >> b)
{
    // a and b have been read
}
else
{
    // a or b could not be read correctly
}
```
Reading from files

- We use objects of type `std::ifstream` to read data from files on disk.
- The three stages for reading from files are similar to those for writing to files:
  - ‘Open’ a specified file for reading (this step fails if the file does not exist)
  - Read data from the file (with input stream syntax, like `std::cin`)
  - ‘Close’ the file
- We will look at the simplest case of reading files of whitespace-separated numbers.
Reading from input file streams

```cpp
#include <fstream>  // for std::ifstream
#include <iostream>  // for std::cout

int main()
{
    double sum = 0.0, d;
    std::ifstream demoFile;  // our file is a std::ifstream

    // try to open a named file for reading
    demoFile.open("input_demonstration.txt");
    // if the file failed to open, return from main with nonzero
    if (!demoFile) return 1;

    // read a number into d. Syntax is like std::cin
    while (demoFile >> d) // loop until file read fails
        sum += d;  // sum up the numbers in the file

    std::cout << "sum = " << sum << std::endl;

    demoFile.close();  // close the file now we are done with it
    return 0;  // return from main with zero (success)
}
```
**Reading from input file streams**

- The statement
  ```java
demoFile.open("inputDemonstration.txt");
```
  reads from the file `inputDemonstration.txt`, which must be in the *working directory*.
  
  - On the command line, the commands `pwd` (Linux/Mac) or `cd` (DOS/Windows) display the working directory.

- The contents of `inputDemonstration.txt` should be a list of floating-point numbers, separated by whitespace (spaces, tabs, newlines), e.g.
  ```
0.5 3 1.75e1
-4.99
  ```

- With this file, the output of the program is
  ```
sum = 16.01
```
Summary

- Flow control (C++ Primer §1.4, Chapter 6)
- Functions (C++ Primer §§7.1, 7.3)
- Global variables
- Input and output streams
- Reading from and writing to files (C++ Primer, §8.4)

Coursework 1:

- Now set, see ‘Coursework & Projects’ section on course website
- Due in 7th October, 5pm, through Blackboard
- **No more than 1–2 pages**
- Can use \LaTeX or Word, and any software for graphs (Excel, MATLAB etc.). See \LaTeX template and instructions on making graphs in MATLAB/Gnuplot on course website.