A more detailed look at C++ syntax

```cpp
#include <iostream>

int main()
{
    double sum = 0.0; // Set up variables
    int nTerms = 100, i=1;

    while (i<=nTerms) /* Loop and sum terms */
    {
        sum += 1.0/(i*i);
        i++;
    }

    std::cout << "Sum of sequence after " << nTerms
             << " terms = " << sum << std::endl; // output result

    return 0;
}
```
#include <iostream>

```cpp
int main()
{
    double sum = 0.0;  // Set up variables
    int nTerms = 100, i=1;

    while (i<=nTerms) /* Loop and sum terms */
    {
        sum += 1.0/(i*i);
        i++;
    }

    std::cout << "Sum of sequence after " << nTerms
               << " terms = " << sum << std::endl;  // output result

    return 0;
}
```
Main function

int main()
{
    // Your code here

    return 0;
}

- The **main** function is called by the operating system when the program starts (it defines the *entry point*).
- All C++ programs must define a **main** function
- For now take the syntax as read; we will look in more detail later at the syntax for functions
#include <iostream>

int main()
{
    double sum = 0.0;  // Set up variables
    int nTerms = 100, i=1;

    while (i<=nTerms) /* Loop and sum terms */
    {
        sum += 1.0/(i*i);
        i++;
    }

    std::cout << "Sum of sequence after " << nTerms << " terms = " << sum << std::endl;  // output result

    return 0;
}
Variables

```c
double sum = 0.0;
int nTerms = 100;
```

- Variables contain data (often numerical), which is modified by the program
- Variables have a *type*, which defines the kind of data stored
  - C++ defines a number of fundamental data types, for integers, fractional values etc. (we have already seen `int` and `double`)
  - The C++ standard library defines many more
  - New types can be defined in a C++ program
  - The type of a variable cannot be changed at run-time
- In C++ we must declare a variable (and its type) before using it.
- Here we are declaring the three variables and *initialising* them using the *literal values* 0.0, 100 and 1.
## Variables — built-in data types

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Description</th>
<th>Literals</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>A text character</td>
<td>'A'</td>
<td>ASCII encoded</td>
</tr>
<tr>
<td>int</td>
<td>4/8</td>
<td>An integer</td>
<td>5</td>
<td>Max. and min. values of at least ( \pm 2 \times 10^9 )</td>
</tr>
<tr>
<td>unsigned</td>
<td>4/8</td>
<td>A natural number</td>
<td>6</td>
<td>Range is 0 to at least ( \pm 4 \times 10^9 )</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>Fractional number</td>
<td>3.142</td>
<td>Floating point, ( \approx 16 ) digits of accuracy, range of between ( \pm (10^{-308} \text{ to } 10^{308}) )</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>Fractional number</td>
<td>3.142f</td>
<td>Floating point, ( \approx 7 ) digits of accuracy</td>
</tr>
<tr>
<td>bool</td>
<td>1</td>
<td>Boolean value</td>
<td>true, false</td>
<td>true = 1, false = 0</td>
</tr>
</tbody>
</table>

...plus various other sizes of integer, and the null void type
Variables — scope

- We may declare variables anywhere in a function before we use them (here at the top of the function).
- The variables are declared inside the `main` function, so they can only be accessed inside this function (this is their scope).
- More generally, variables will be localised to the `block` (delimited by braces `{}`) in which they are declared.
- What are the values of `i`, `j` and `k` after each line?

```c
int i = 0; // ?
int j = 1; // ?
{
    int i = 2; // ?
    int k = 3; // ?
}
// ?
```
Variables — scope

- We may declare variables anywhere in a function before we use them (here at the top of the function).
- The variables are declared inside the main function, so they can only be accessed inside this function (this is their scope).
- More generally, variables will be localised to the block (delimited by braces {}) in which they are declared.
- What are the values of $i$, $j$ and $k$ after each line?

```c
int i = 0; // i=0; j, k undefined
int j = 1; // i=0; j=1; k undefined
{
    int i = 2; // i=2; j=1; k undefined
    int k = 3; // i=2; j=1; k=3
}
// i=0; j=1; k undefined
```
#include <iostream>

int main()
{
    double sum = 0.0; // Set up variables
    int nTerms = 100, i=1;

    while (i<=nTerms) /* Loop and sum terms */
    {
        sum += 1.0/(i*i);
        i++;
    }

    std::cout << "Sum of sequence after " << nTerms
              << " terms = " << sum << std::endl; // output result

    return 0;
}
Expressions

Expressions are statements that typically perform calculations.

They are expressed in a mathematical syntax with:

- **Operators**, such as `+`, `-`, `*`, `=`, `==`, `+=`, `<<`, `++` (~50 in total)
- **Operands**, which are variables (e.g. `i`, `sum`) and literals (e.g. `1.0`, "Text")

- Operators have a *precedence* which determined the order they are evaluated (an extension of the familiar ‘BODMAS’ rule)
- Operators always **evaluate to** a value (1 + 2 evaluates to 3)
- Operators can also **change** the value of their operands:
  - `a = b` changes the value of `a` to `b` (as well as evaluating to `b`)
Expressions

How do we evaluate the following expression?

```c
sum += 1.0/(i*i);
```

- The operators (in order of precedence) are (), /, *, +=
- The highest-precedence operator ( ) returns the value of the expression inside it, $i*i$, which we now evaluate:
  - The operator * returns the product of its two operands (here $i^2$)
- The next highest-precedence operator is /, which returns its left operand ($1.0$) divided by its right ($i*i$)
- The operator += increments its left operand (sum) by its right,

$$sum \leftarrow sum + \frac{1.0}{i^2}$$
# Expressions – More operators

Some common operators, in order of decreasing precedence:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Evaluates to:</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a++, a--</td>
<td>a, <strong>and sets a to a+1, a−1</strong></td>
<td>postfix increment/decrement</td>
</tr>
<tr>
<td>++a, --a</td>
<td>a+1, a−1, <strong>and sets a to a+1, a−1</strong></td>
<td>prefix increment/decrement</td>
</tr>
<tr>
<td>!a</td>
<td>true iff a is false</td>
<td>logical NOT</td>
</tr>
<tr>
<td>*a, &amp;a</td>
<td>value-at address, address-of variable</td>
<td>dereference, address-of</td>
</tr>
<tr>
<td>*, /, %</td>
<td>a × b, a/b, a mod b</td>
<td>multiply, divide, remainder</td>
</tr>
<tr>
<td>a+b, a−b</td>
<td>a − b, a + b</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>true iff a &lt; b, a &gt; b, a ≤ b, a ≥ b</td>
<td>&lt;, &gt;, ≤, ≥</td>
</tr>
<tr>
<td>a == b</td>
<td>true iff a equal to b</td>
<td>equality</td>
</tr>
<tr>
<td>a != b</td>
<td>true iff a ≠ b</td>
<td>inequality</td>
</tr>
<tr>
<td>a &amp;&amp; b</td>
<td>true iff a and b true</td>
<td>logical AND</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>a = b</td>
<td>b, <strong>and sets a to b</strong></td>
<td>assignment</td>
</tr>
<tr>
<td>a += b</td>
<td>a+b, <strong>and sets a to a+b</strong></td>
<td>assignment by sum</td>
</tr>
<tr>
<td>(=, -=, *=, /=, %=, similarly)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(See [http://tinyurl.com/operator-precedence](http://tinyurl.com/operator-precedence) for a complete list)
Expressions – Equality

Be careful of the two ‘equals’ operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Evaluates to:</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
<td>true iff a equal to b</td>
<td>equality</td>
</tr>
<tr>
<td>a = b</td>
<td>b, and sets a to b</td>
<td>assignment</td>
</tr>
</tbody>
</table>

- Is \( a = b \) equivalent to \( b = a \)?
- Is \( a == b \) equivalent to \( b == a \)?
#include <iostream>

int main()
{
    double sum = 0.0;  // Set up variables
    int nTerms = 100, i=1;

    while (i<=nTerms) /* Loop and sum terms */
    {
        sum += 1.0/(i*i);
        i++;
    }

    std::cout << "Sum of sequence after " << nTerms << " terms = " << sum << std::endl;  // output result

    return 0;
}
Displaying text with streams

std::cout << "Sum of sequence after " << nTerms << " terms = " << sum << std::endl;

- To display text to the screen, we send it to the std::cout object using the insertion operator <<.
- We can send text strings in quotes ("text") and variables of many types including int (nTerms) and double (sum)
- To move onto the next line of text, we sent another built-in object, std::endl
- The code above displays the text:

  Sum of sequence after 100 terms = 1.63498

- Note: To use std::cout and std::endl we must include the appropriate library
```cpp
#include <iostream>

int main()
{
    double sum = 0.0;  // Set up variables
    int nTerms = 100, i=1;

    while (i<=nTerms) /* Loop and sum terms */
    {
        sum += 1.0/(i*i);
        i++;
    }

    std::cout << "Sum of sequence after " << nTerms
                << " terms = " << sum << std::endl; // output result

    return 0;
}
```
# Libraries

```cpp
#include <iostream>
```

- `std::cout` and `std::endl`, along with many other objects for common tasks, are provided by the C++ *standard library*.
- We include a part of this library with the code
  ```cpp
  #include <header_name>  // typically at the top of our code
  ```
- Standard library *headers* include:
  - `<iostream>` (Displaying text on the screen, keyboard input)
  - `<fstream>` (Loading and saving files to disk)
  - `<vector>`, `<list>`, `<map>` etc. (Data structures)
  - `<string>` (Text strings)
  - `<cmath>` (Common mathematical functions, e.g. $\sqrt{x}$)
- There are many third-party C++ libraries that implement more specialised algorithms – or you can write your own.
#include <iostream>

int main()
{
    double sum = 0.0; // Set up variables
    int nTerms = 100, i=1;

    while (i<=nTerms) /* Loop and sum terms */
    {
        sum += 1.0/(i*i);
        i++;  
    }

    std::cout << "Sum of sequence after " << nTerms
    << " terms = " << sum << std::endl; // output result

    return 0;
}
While loops

To execute statements many times, we can use a loop.

C++ supports a number of loop constructs, including while, for and do while.

The syntax of a while loop is

```
int i=1;
while (i<=nTerms)
{
   // ...
   i++;
}
```
While loops

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

```
while ([condition])
{
    [statements]
}
```

- `[statements]` represents the list of statements to be executed repeatedly.
- This is known as the *loop body*
While loops

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

- `[condition]` is an expression evaluated before before the loop body each time it is repeated.
- The loop body is executed again only if the expression evaluates to true.
- If the condition is false, execution jumps to the first statement after the while loop.
While loops – some examples

- To loop a fixed number of times, we can set up a counter variable:

```c
int i = 1, nTerms = 100;
while (i <= nTerms)
{
    // do useful work here
    i++;
}
```

- This counter can work in a number of ways:

```c
int i = 100; // start from maximum value, count down
while (i > 0)
{
    // do useful work here
    i--;
}
```

- Next lecture we will see how to do this in a more compact way, using `for` loops.
Summary

• Fundamentals of C++
  • Variables (declaration, types)
  • Expressions (operators, precedence)
  • Displaying text (std::cout)
  • Libraries (#include <...>)
  • While loops
Before next time:

1. Try installing a C++ compiler on your own computer (see ‘Guide to Compiling C++’ on course website)
   - Free compilers available for Windows/Mac/Linux

2. Look at C++ Primer (Fourth Edition) by Lippman et al.
   - Available free as an e-book (on campus, or log in) at http://proquestcombo.safaribooksonline.com/0201721481
   - Read sections 6.1–6.12

3. Computer lab Tuesday 4pm–5.50pm, Alan Turing Building G.105