Introduction to Scientific Programming with C++

Session 0: Basics

Martin Uhrin

UCL

February 11-13th 2013

Table of Contents

Introduction to C++

A little history

- Created in 1979 as an extension of C by this guy:

Introduction to C++

A little history

- Created in 1979 as an extension of C by this guy:

Figure: Bjarne Stroustrup, creator of C++.

Introduction to C++

A little history

- Created in 1979 as an extension of C by this guy:

Figure: Bjarne Stroustrup, creator of C++.

- Built to be fast *and* scalable and so has a mix of high and low-level constructs.

Hello World!

Where it all begins

```cpp
// My first C++ program

#include <iostream>

int main()
{
  std::cout << "Hello world!";
  return 0;
}
```

../code/0_basics/lectures/hello_world.cpp

Output: `Hello world!`

Hello World dissection

```
// My first C++ program
```

Hello World dissection

```
// My first C++ program

#include <iostream>
```

Hello World dissection

```
// My first C++ program

#include <iostream>

int main()
```

Hello World dissection

```
// My first C++ program

#include <iostream>

int main()
{


}
```

Hello World dissection

```
// My first C++ program

#include <iostream>

int main()
{
        cout << "Hello World!";

}
```

Hello World dissection

```
// My first C++ program

#include <iostream>

int main()
{
  std::cout << "Hello World!";

}
```

Hello World dissection

```
// My first C++ program

#include <iostream>

int main()
{
  std::cout << "Hello World!";
  return 0;
}
```

Going from source code to an executable

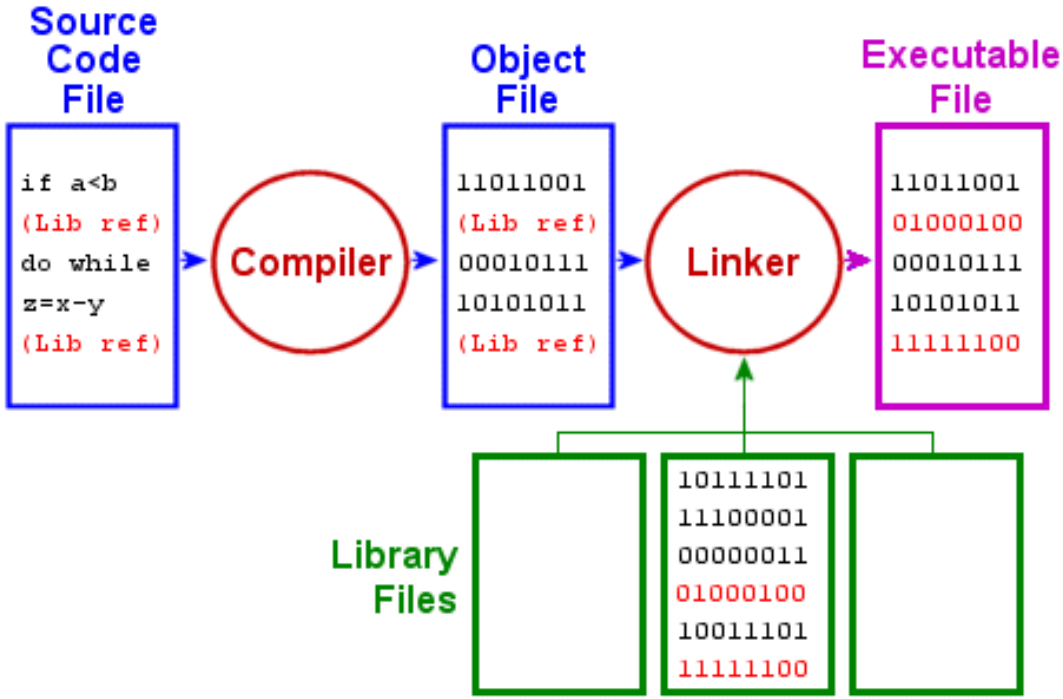

Figure: Compilation and linking of a C++ program[1].

[1] Source `http://www.aboutdebian.com/compile.htm` © Keith Parkansky

Going from source code to an executable

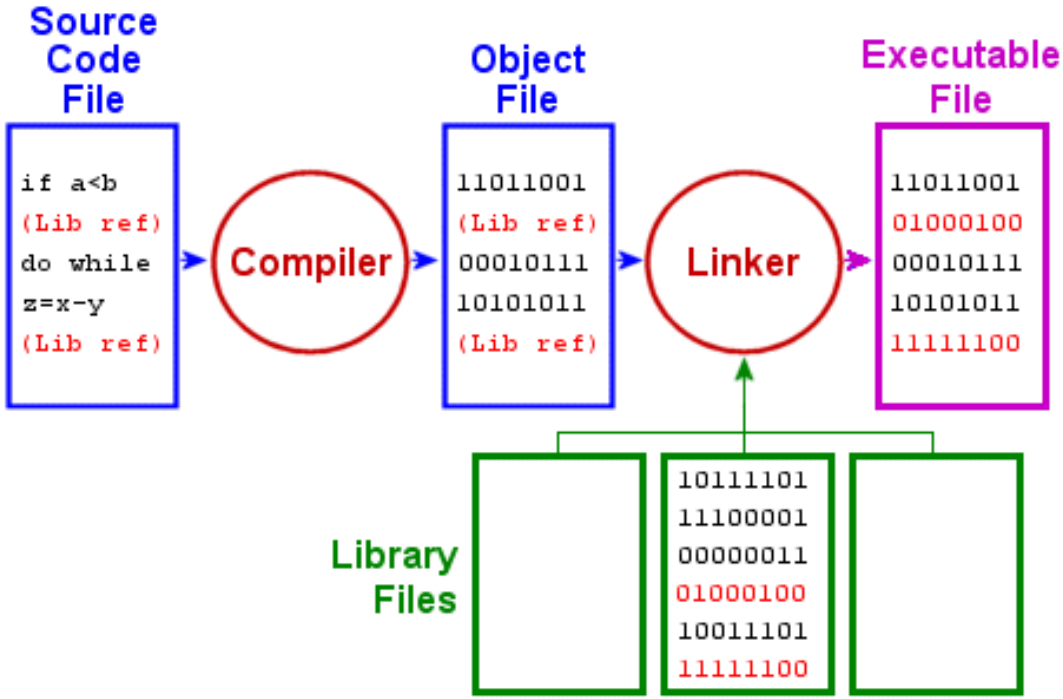

Figure: Compilation and linking of a C++ program[1].

There are range of compilers, some common ones are *g++* (GNU), *icc* (Intel), *pgicc* (PGI) and others.

[1] Source `http://www.aboutdebian.com/compile.htm` © Keith Parkansky

Whitespace

Definition

whitespace spaces, tabs, and (sometimes) new lines.

Whitespace

Definition

whitespace spaces, tabs, and (sometimes) new lines.

Completely equivalent as far as compiler is concerned:

```
#include <iostream>
int main(){std::cout<<"Hello world!";return 0;}
```

```
#include <iostream>
int main()
{
std::cout
<<
"Hello world!"
;
return
0
;
}
```

Case sensitivity

C++ is case sensitive! This means that:

```
int main()
```

is different from

```
INT MAIN ()
```

which is different from

```
int Main()
```

and only the first version is correct.

Comments

Two types of comment can be used:

```
// This is a line comment.
// It ends at the end of the line.
// int myVariable = 0; This code will not execute!
```

```
/* This is a C-style comment.
   It ends when the closing star-slash is reached. */
```

Use comments liberally - they are enormously useful!

Do

Avoid stating the obvious. A good comment will not say *what* is happening but rather *why*.

Defining a variable

Definition

variable A named portion of memory used to store a determined value.

Let's define some variables:

```
int anInteger;
double aDouble;
unsigned short i;
float x, y, z;
```

Format: `variable_type variable_name, variable_name2;`

Naming variables

A variable name is an example of an identifier.

Definition

identifier an identifier is a sequence of characters used to denote the name of a variable, function[2], class[2] or any entity you need to refer to in your code.

[2] We'll see what these are shortly.

[3] See `http://en.cppreference.com/w/cpp/keyword` for full list.

Naming variables

A variable name is an example of an identifier.

Definition

identifier an identifier is a sequence of characters used to denote the name of a variable, function[2], class[2] or any entity you need to refer to in your code.

Identifiers can be any sequence of letters, digits or underscore characters but they must *not*:

- start with a digit,
- be one of the reserved keywords[3].

[2] We'll see what these are shortly.

[3] See `http://en.cppreference.com/w/cpp/keyword` for full list.

Naming variables

A variable name is an example of an identifier.

Definition

identifier an identifier is a sequence of characters used to denote the name of a variable, function[2], class[2] or any entity you need to refer to in your code.

Identifiers can be any sequence of letters, digits or underscore characters but they must *not*:

- start with a digit,
- be one of the reserved keywords[3].

Definition

keyword a word that has a special meaning in the C++ language.

[2] We'll see what these are shortly.

[3] See `http://en.cppreference.com/w/cpp/keyword` for full list.

Naming variables

Much like naming children

Do

- Give variables meaningful names, even if this means more typing!
  Good: `daysOfWeek`, `sumSq`, `isEnabled`, `unitCell`
  Would the variable name make sense in that context when looking at the code a year later?

Naming variables

Much like naming children

Do

- Give variables meaningful names, even if this means more typing!
  Good: `daysOfWeek`, `sumSq`, `isEnabled`, `unitCell`
  Would the variable name make sense in that context when looking at the code a year later?
- Use variable names, much like comment, to convey intent, e.g.:

```
double rootMeanSquare; // .. go on to
    calculate rms
```

  This will make it obvious to the user that the code after this variable should calculate the rms and store it in this variable.

Naming variables

Much like naming children

Do

- Give variables meaningful names, even if this means more typing!
  Good: `daysOfWeek`, `sumSq`, `isEnabled`, `unitCell`
  Would the variable name make sense in that context when looking at the code a year later?
- Use variable names, much like comment, to convey intent, e.g.:

```
double rootMeanSquare; // .. go on to
    calculate rms
```

  This will make it obvious to the user that the code after this variable should calculate the rms and store it in this variable.

Don't

Use abbreviations unless they're VERY common.
Use bad names: `data`, `dRange`, `a`, `ccn`, `value`, `one`

Variable types

Fundamental data types:

Type	Size	Values
`bool`	1 byte	true or false
`int`	4 bytes	-2,147,483,648 to 2,147,483,647
`double`	8 bytes	2.2e-308 to 1.8e308

Variable types

Fundamental data types:

Type	Size	Values
`bool`	1 byte	true or false
`int`	4 bytes	-2,147,483,648 to 2,147,483,647
`double`	8 bytes	2.2e-308 to 1.8e308
`char`	1 byte	256 character values
`unsigned short int`	2 bytes	0 to 65,353
`short int`	2 bytes	-32,768 to 32,767
`unsigned int`	4 bytes	0 to 4,294,967,295
`unsigned long int`	8 bytes	0 to 18,446,744,073,709,551,615
`long int`	8 bytes	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807
`float`	4 bytes	1.2e-38 to 3.4e38

Using variables

```cpp
#include <iostream>

int main()
{
  double G = 6.6738e-11;
  double massOfEarth = 5.9722e24;
  double massOfMoon = 7.3477e22;
  double r = 384400e3;
  double force;

  force = G * massOfEarth * massOfMoon / (r * r);

  std::cout << "Force between Earth and Moon is: "
            << force << "\n";

  return 0;
}
```

../code/0_basics/lectures/force_calc.cpp

Pretty easy, right?

Constants

Spot the difference:

```cpp
#include <iostream>

int main()
{
  const double G = 6.6738e-11;
  const double massOfEarth = 5.9722e24;
  const double massOfMoon = 7.3477e22;
  const double r = 384400e3;
  double force;

  force = G * massOfEarth * massOfMoon / (r * r);

  std::cout << "Force between Earth and Moon is: " << force << "\n";

  G = 7e-11; // WON'T COMPILE. WHAT KIND OF A UNIVERSE WOULD WE BE LIVING IN
             // IF G COULD VARY??

  return 0;
}
```

../code/0_basics/lectures/force_calc_const.cpp

Constants

Spot the difference:

```
#include <iostream>

int main()
{
  const double G = 6.6738e-11;
  const double massOfEarth = 5.9722e24;
  const double massOfMoon = 7.3477e22;
  const double r = 384400e3;
  double force;

  force = G * massOfEarth * massOfMoon / (r * r);

  std::cout << "Force between Earth and Moon is: " << force << "\n";

  G = 7e-11; // WON'T COMPILE. WHAT KIND OF A UNIVERSE WOULD WE BE LIVING IN
             // IF G COULD VARY??

  return 0;
}
```

../code/0_basics/lectures/force_calc_const.cpp

Do

Make everything `const` unless it absolutely has to vary. This has benefits for both the speed and correctness of your code.

Scope

Scope this out

Definition

scope the region of a program that a variable can be used in. This determines the "life-time" of a variable.

The scope of a variable is defined by the block, formed by braces {}, within which it is declared.

[4] Source: `http://www.cplusplus.com/doc/tutorial/variables/`.

Scope

Scope this out

Definition

scope the region of a program that a variable can be used in. This determines the "life-time" of a variable.

The scope of a variable is defined by the block, formed by braces {}, within which it is declared.

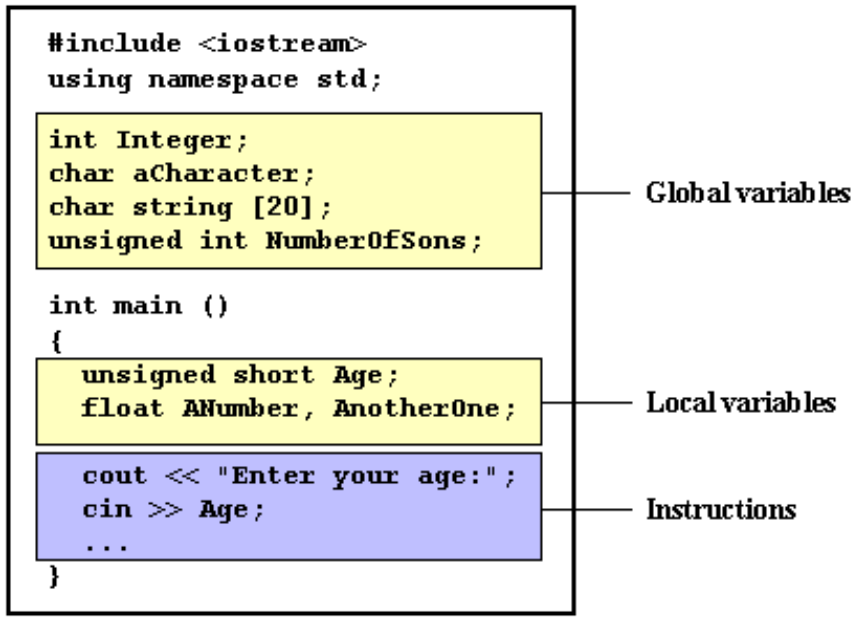

Figure: Variable scope[4].

[4] Source: http://www.cplusplus.com/doc/tutorial/variables/.

Scope

Scope this out

Definition

scope the region of a program that a variable can be used in. This determines the "life-time" of a variable.

The scope of a variable is defined by the block, formed by braces {}, within which it is declared.

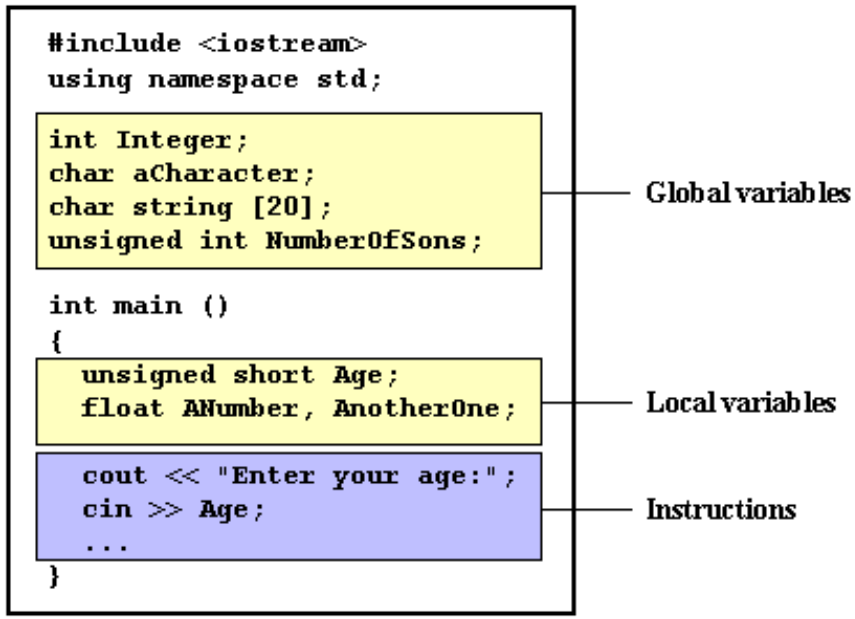

Figure: Variable scope[4].

Don't

Declare variables in global scope. Limit the scope as much as possible to reduce chance of side-effects. Global constants, however, are fine.

[4] Source: `http://www.cplusplus.com/doc/tutorial/variables/`.

Simple operators

Assignment: =

```
a = 5; a = b; a = b = c;
```

Simple operators

Assignment: =

```
a = 5; a = b; a = b = c;
```

Arithmetic operators: +, -, *, /, %

All obvious with the exception of the *modulo* operator (%). This gives the remainder after division e.g.

```
a = 22 % 7;
```

will set a to 1 as $\frac{22}{7} = 3 \times 7$ remainder 1.

Simple operators

Assignment: =

```
a = 5; a = b; a = b = c;
```

Arithmetic operators: +, -, *, /, %

All obvious with the exception of the *modulo* operator (%). This gives the remainder after division e.g.

```
a = 22 % 7;
```

will set a to 1 as $\frac{22}{7} = 3 \times 7$ remainder 1.

Compound assignment: +=, -=, *=, /=, %=

Perform the operation on the current value of the variable and then set it to the new value, e.g.:

```
a += 5; a *= b;
```

Integer arithmetic

Warning!

In C++ integer arithmetic truncates (effectively rounds down):

```
int dividend = 20, divisor = 7;
int someInteger = dividend / divisor; // = 2
```

Storing the result in a double doesn't help as the arithmetic has already been done:

```
double someDouble = dividend / divisor; // = 2
```

What gives?

Integer arithmetic

Warning!

In C++ integer arithmetic truncates (effectively rounds down):

```
int dividend = 20, divisor = 7;
int someInteger = dividend / divisor; // = 2
```

Storing the result in a double doesn't help as the arithmetic has already been done:

```
double someDouble = dividend / divisor; // = 2
```

What gives?

To get around we can "cast" the variable to another type:

1. ```
   someDouble = static_cast<double>(dividend) /
       static_cast<double>(divisor); // = 2.85714
   ```
   ```

Unary operators: `++`, `--`

```
a++;
```

This is equivalent to:

```
a += 1;
```

Unary operators: ++, --

```
a++;
```

This is equivalent to:

```
a += 1;
```

Relational and equality operators: ==, !=, >, <, >=, <=

Evaluates to a boolean (true or false) value e.g.:

```
bool areNotEqual = (a != b);
```

Unary operators: ++, --

```
a++;
```

This is equivalent to:

```
a += 1;
```

Relational and equality operators: ==, !=, >, <, >=, <=

Evaluates to a boolean (true or false) value e.g.:

```
bool areNotEqual = (a != b);
```

Don't

Mix up = and == this will cause endless headaches! Consider:

```
a = 5; b = 6; areEqual = (a = b);
```

This is a problem because C++ considers any number other than 0 be true!

Logical operators: `!`, `&&`, `||`

These act on boolean values in the following ways:

NOT

a	!a
`true`	`false`
`false`	`true`

AND

a	b	a && b
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

OR

a	b	a || b
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

Logical operators: `!`, `&&`, `||`

These act on boolean values in the following ways:

NOT

a	!a
`true`	`false`
`false`	`true`

AND

a	b	a && b
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

OR

a	b	a || b
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

Do

Keep it simple: don't try and do too much in a single line. While this:

```
result = (i < 10) && (++i < n);
```

is a valid expression, deciphering what it does is a lot of work. Instead use:

```
result = i < 10;
++i;
result = result && i < n;
```

Operator precedence

No, you got first...

Precedence	Op.	Associativity
1	++ -- !	Left to Right
2	* / %	Left to Right
3	+ -	
4	< <= > >=	
5	== !=	
6	&&	
7	||	
8	=	Right to Left

Operator precedence tells you the order that an expression will be evaluated in. Some are obvious but consider:

```
a = 21 + 7 % 2;
```

which could be interpreted as

```
a = 21 + (7 % 2);
    // this
a = (21 + 7) % 2;
    // or this.
```

In fact the first version is correct.

Operator precedence

No, you got first...

Precedence	Op.	Associativity
1	`++ --` `!`	Left to Right
2	`* / %`	Left to Right
3	`+ -`	
4	`< <=` `> >=`	
5	`== !=`	
6	`&&`	
7	`||`	
8	`=`	Right to Left

Operator precedence tells you the order that an expression will be evaluated in. Some are obvious but consider:

```
a = 21 + 7 % 2;
```

which could be interpreted as

```
a = 21 + (7 % 2);
    // this
a = (21 + 7) % 2;
    // or this.
```

In fact the first version is correct.

Do

Use parentheses to make an expressions more clear even if they are not necessary.

Operator precedence

No, you got first...

Precedence	Op.	Associativity
1	`++ --` `!`	Left to Right
2	`* / %`	Left to Right
3	`+ -`	
4	`< <=` `> >=`	
5	`== !=`	
6	`&&`	
7	`||`	
8	`=`	Right to Left

Operators that have the same precedence are evaluated according to their *associativity* e.g.:

```
a = b = c;
    //
    evaluates
    as
a = (b = c);
```

because = is right to left associative.

Operator precedence

No, you got first...

Precedence	Op.	Associativity
1	++ -- !	Left to Right
2	* / %	Left to Right
3	+ -	
4	< <= > >=	
5	== !=	
6	&&	
7	||	
8	=	Right to Left

Operators that have the same precedence are evaluated according to their *associativity* e.g.:

```
a = b = c;
    //
    evaluates
    as
a = (b = c);
```

because = is right to left associative. While:

```
a * b / c;
//evaluates as
(a * b) / c;
```

are left to right associative.

Standard output (cout)

You've already met `cout`, it uses the *indirection operator* (<<) to print to the screen e.g.:

```
std::cout << "Have some pi: ";
std::cout << 3.1415926;
std::cout << a;
```

Standard output (cout)

You've already met `cout`, it uses the *indirection operator* (<<) to print to the screen e.g.:

```
std::cout << "Have some pi: ";
std::cout << 3.1415926;
std::cout << a;
```

We can use << more than once in the same statement:

```
double t0 = 1.5, t1 = 2.5;
cout << "t0: " << t0 << ", t1: " << t1
     << ", delta: " << t1 - t0;
```

Output: `t0:  1.5, t1:  2.5, delta:  1`

Standard output (cout)

You've already met `cout`, it uses the *indirection operator* (<<) to print to the screen e.g.:

```
std::cout << "Have some pi: ";
std::cout << 3.1415926;
std::cout << a;
```

We can use << more than once in the same statement:

```
double t0 = 1.5, t1 = 2.5;
cout << "t0: " << t0 << ", t1: " << t1
     << ", delta: " << t1 - t0;
```

Output: `t0: 1.5, t1: 2.5, delta: 1`

If you want a new line you have to use `\n`:

```
double t0 = 1.5, t1 = 2.5;
cout << "t0: " << t0 << "\nt1: " << t1 << "\n";
```

Output:
```
t0: 1.5
t1: 2.5
```

Standard input (`cin`)

Extracting information out of the user

To get input from the user, use the extraction operator (>>) of the `cin` object (pronounced *see-in*) e.g.:

```
double radius;
std::cin >> radius;
```

at this point the program will stop and wait for the user to enter a number and push RETURN.

Standard input (`cin`)

Extracting information out of the user

To get input from the user, use the extraction operator (>>) of the `cin` object (pronounced *see-in*) e.g.:

```
double radius;
std::cin >> radius;
```

at this point the program will stop and wait for the user to enter a number and push RETURN. As with output we can use >> more than once in a statement e.g.:

```
std::cin >> width >> height;
```

in this case the program will wait for two sets of numbers to be entered. They can be separated by a space, tab or a newline.

Complete example

Putting it all together

```
#include <iostream>

int main()
{
  unsigned int width, height;
  std::cout << "Please enter a width and height: ";
  std::cin >> width >> height;
  std::cout << "Area is: " << width * height << "\n";

  const double ratio = static_cast<double>(height) /
    static_cast<double>(width);
  std::cout << "Ratio is: 1:" << ratio
            << " (width:height)" << "\n";

  const bool isSquare = (width == height);
  std::cout << "Is it a square? " << isSquare << "\n";

  return 0;
}
```

../code/0_basics/lectures/rect_info.cpp