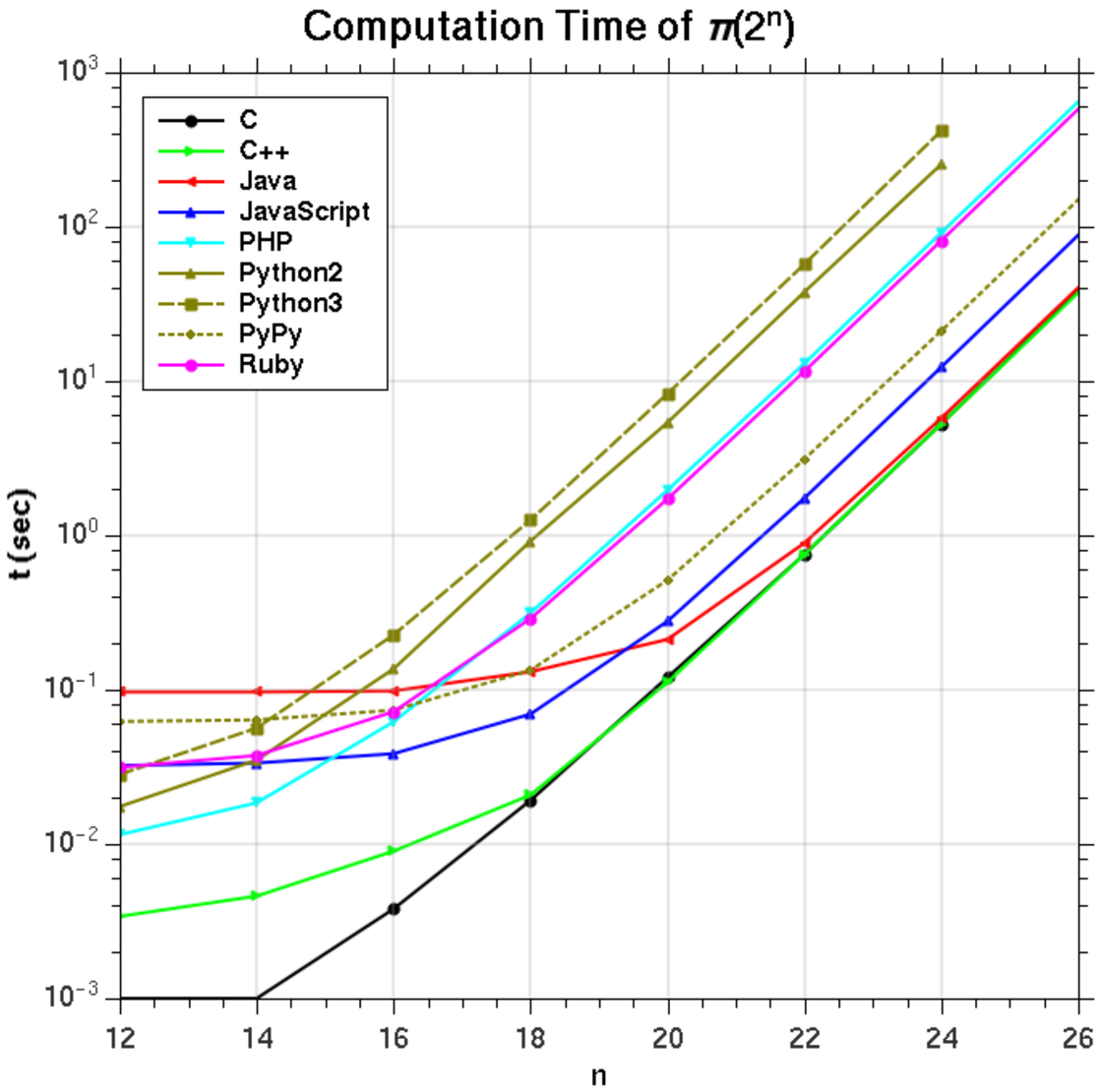
C++ For Researchers

Warwick RSE

21/03/2024

Why Program in C++



- Faster code than Matlab, Python, R etc.
- As fast as C (at least on any problem that isn't already very fast)
- Easier to write than C and fewer opportunities for serious errors
  - Writing C like code in C++ is not better!
- Lots of libraries and good to write your own libraries in
- Lots of experienced developers in academia
- Not as good with arrays as Fortran, but many other advantages

Language Standards

- C++ is a language that is defined by ISO standards
  - Mostly a kind of “tick-tock” development of major change followed by update based on experience
- C++98/03 - Original C++ standard. Introduced much of the “shape” of the language. 03 firmed up the structure and approach
- C++11/14 - Added new features that were aimed at improving developer productivity. 14 expanded the capabilities of the new features
- C++17 - Mostly aimed at adding features to make C++ code more readable and remove boilerplate code
- C++20 - Variety of changes, mostly concentrating on features for people writing libraries in C++
- C++23 - Latest standard, bit of a mixture of things again

Text and whitespace

- C++ is a **case sensitive language**, so when anything is used it must match the case when it was defined. Only use this to ensure consistent style **DO NOT WRITE CODE WHERE TWO THINGS HAVE THE SAME NAME APART FROM CASE!**

- White space is not used in C++

  - Lines of code are ended with **;** not with line ends

  - Line indentation is only there to aid humans reading the code

  - Blocks of code are enclosed in pairs of **{ }**

Function calling

- When you call a function in a language the values that you give the function have to get into the function somehow
- Two general approaches
  - Pass by value - copy the values into the function. Changing that copy doesn't change the original value (C/C++)
  - Pass by reference - the function is given a reference to the value - changing the value inside the function changes the original value (Fortran)
    - Slight wrinkle if you pass the same variable to two arguments (don't do this)

Compiling C++

Compiling C++

- C++ is a **compiled** language
  - The source code has to be run through a compiler before it can be executed
- If you are familiar with Fortran or C then the compile lines for C++ are very similar, just change the compiler name
  - g++ - GNU
  - clang++ - Clang/LLVM
  - icpc (classic), icpx(modern/OneAPI) - Intel
  - Many others but these are the ones we use in SCRTP
- C++ files have the **.cpp** extension by convention and C++ compilers sometimes won't compile programs without this extension
  - There are also **header files** that have the **.h** or **.hpp** extension

Compiling C++

- Compilers always try and make your code faster, by **optimisation**
  - You have a choice of how much optimisation is applied but there is always some (for most codes -O3 is most optimised and -O0 is least optimised)
- Modern compilers perform “as if” compilation
- They produce code that gives the same effects “as if” they had run your code, but otherwise don’t have to have any particular connection to your code at all
- This is why language standards are important - if you write code that is not permitted (“undefined behaviour”) then the compiler can chose to do literally anything that it likes when it encounters that code!
- Less serious is “unspecified behaviour” where the language standard doesn’t say what a compiler should do, but it shouldn’t do anything silly
- Finally, there is also “implementation defined” behaviour that is guaranteed to not to do anything crazy and will always do the same thing, but will be different on different compilers
- Don’t invoke any of these

Compiling C++

- Single line compilation
  - g++ file1.cpp file2.cpp file3.cpp -o output_prog
  - The files do not have to be in any particular order (unlike some other languages)
  - Final executable program is “output_prog” and can just be run as ./output_prog
- Usually want to specify -O3 as a command line option to the compiler to tell the compiler to optimise the code as much as possible
- Note that there are no .h or .hpp files in the compile line
  - Header files are always **included** inside cpp files and the cpp files are compiled

Compiling C++

- Multi line compilation
  - g++ -c file1.cpp
    g++ -c file2.cpp
    g++ -c file3.cpp
    g++ file1.o file2.o file3.o -o output_prog
  - First lines create .o intermediate files from the .cpp files
    - Technically these are **compile lines**
  - Final line builds “output_prog” again from those files. The .o files don’t need to be in any particular order on the compile line
    - Technically this is a **build** or **link** line
  - If you want optimisation then you **have** to add the -O3 flag to the compile lines. Most optimisation flags do nothing on the link line, but are harmless there. Some flags (like -flto in g++) need to be specified on both the link and compile lines

Hello World

Boilerplate code

```
int main(){

}
```

- This is the simplest valid C++ code - it compiles and runs, but does absolutely nothing
- It shows the concept of the **entry point** a location that the operating system calls in order to start your program
- In C++ (and C) it is a function called **main**
  - Note that after **main** there are curly brackets - these mark out the extent of the function
  - This is only one form of **main** that exists, the others take parameters for reading the command line parameters that you pass to the code
- The equivalent in Fortran is the **PROGRAM** block, and many interpreted or JIT compiled languages just execute any code that is not inside a function when the program starts

Hello World

```
#include <iostream>
int main(){
 std::cout << "Hello world\n";
 std::cout << "Hello " << "again "
    << " world\n";
}
```

- This is a program that actually does something!
- It shows a few important features of C++

1. Most of C++ is not just naturally available, you have to **include** the right header file, here **iostream** for printing
2. All of the lines end with a **;**
3. std::cout is the printing function in C++ and you output strings to it using the **<<** (insertion) operator
4. C++ does not automatically end a line after printing something we have to put in “\n” to end the line manually
5. You can chain the insertion operator together to print many things on a single line
6. I have split the printing line over two lines and have indented the second line relative to the first. That is OK, the compiler only uses the **;** to determine a line ending. The indent is just to help a reader

Variable Creation

Variables

- All variables in C++ have to be **declared** (i.e. give a type) before they can be **defined** (i.e given a value)
- Every variable has a type that has to be known before the code can be compiled
  - You can under some circumstances say to the compiler “You know what type this has to be for the code to compile, so make it the right type” and we will cover this later
- This applies to “variable-like” things as well as variables
  - So parameters to functions have to have types
  - The value returned from a function has to have a type
  - All of the elements of structures have to have types
- Type checking can actually be very helpful for ensuring that your code is correct, but it is also needed for performance - a lot of time is spent in languages like Python checking what type something actually is so that the CPU can work with it

Variables

```
#include <iostream>
#include <string>

int main(){


   int i;
   int j,k;
   float x = 1.0;
   double y = 1.234;
   std::string s = "Hello world!";
   std::string other_s;

}
```

- This is back to a program that doesn't print anything, but now it creates variables
- You start a line with the type of the variable that you are declaring and the you put one or more variable names separated by commas.
- Note that the std::string type needs you to include the **string** header before you can use it
- You can optionally give a variable an initial value when you create it (there are actually several ways, but here we are using = because it looks most natural)
  - Some variable types in C++ **must** be given an initial value
- There are some special variable types

Const variables

- Inherited from C, but more commonly used and extended in C++
- You can flag variables as **constant** by putting **const** before the type of the variable when you declare it
- **const** variables **must** be assigned a value on the line where they are declared
- After that line they **cannot** be changed
  - If you try to change a **const** variable's value the code will not compile
- **const** parameters to functions cannot be changed inside that function
- You cannot pass a **const** value to a function as a non-**const** parameter

auto variables

- Quite often in C++ you will see code like
  - **int value = get_integer();**
- Fine for integer, but for more complex functions you have to find out and match the variable type to the return type
  - The compiler checks that you've used the right type, so it must know what the right type is - can't it work out the type of my variable for me?
- **YES** - use **auto** in place of the type when declaring a variable and assign it a value on the same line and the compiler will "paste-in" the correct return type
- You are still giving the variable a fixed type, just telling the compiler to work out which

Loops

Loops

```
#include <iostream>

int main(){
  for (int i=0;i<10;i++){
    std::cout << "Hello from loop iteration “
     << i << "\n";
  }
}
```

- Loops are the same idea in every programming language
  - Do the same operation multiple times
- C++ has several different types of loop but by far the most common is the simple **for loop**
- In this type of loop you have a block surrounded by **{ }** and started with a **for** command

Loops

```
#include <iostream>

int main(){
  for (int i=0;i<10;i++){
    std::cout << "Hello from loop iteration “
     << i << "\n";
  }
}
```

- The **for** command has three parts inside it separated with ;
- The first part runs only once, when the loop starts and initialises the loop variable.
  - Here we are both declaring the loop variable and giving it an initial value
  - This is useful because one only rarely wants the value of a loop variable when the loop is finished and this constrains it to only exist while the loop is running

Loops

```
#include <iostream>

int main(){
  for (int i=0;i<10;i++){
    std::cout << "Hello from loop iteration “
     << i << "\n";
  }
}
```

- The second section is run at the start of every iteration of the loop
- It is a test to see if the loop is finished or if it should continue
- Here we are testing for if “i” is less than 10
- The loop will run from 0 to 10 inclusive
- This can be any test that you like
- It must evaluate to a value that can be interpreted as true or false

Loops

```
#include <iostream>

int main(){
  for (int i=0;i<10;i++){
    std::cout << "Hello from loop iteration “
     << i << "\n";
  }
}
```

- The final section runs at the end of each iteration of the loop
- It is generally used to increment the loop counter, but can be used to do anything that you want
- Here we just increment i using the increment operator ++

Loops

- The example pack includes other types of loops
- The **while** loop that executes until a condition is false at the start of an iteration
- The **do while** loop that executes until a condition is false at the end of an iteration
  - This is an important difference because these loops are guaranteed to run once
- There are also other variations on the for loop that we mention in the example pack

Conditionals

Conditionals

```
#include <iostream>

int main(){

   int candidate=1;
   int test=4;

   if (test>candidate) std::cout <<
    "Test is greater than candidate\n";
}
```

- **Conditionals** (also called **if statements** after the most common keyword implementing them) mean that code only runs if a certain condition is met
- In C++ they are very simple just
  - if (condition) code_to_run_if_condition_true
- In the example to the left the condition is the test value being greater than the candidate value
- The strange line breaks on the if line are just to make them fit on the slide - look in the example pack

Conditionals

```
#include <iostream>

int main(){

  int candidate=1;
  int test=4;

 if (test>candidate) {
    std::cout << "Test is greater than candidate\n";
  } else {
    std::cout <<
     "Test is less than or equal to candidate\n";
  }
}
```

- This conditional shows two features

  1. After the end of the **if** statement itself put **{ }**. Anything between those braces will only execute if the condition is true

  2. After the close **}** put **else** and then another **{ }** set. Anything in the set of **{ }** after the else will run if the initial condition is false

- If you want to then you can put another if statement after the else (before the **{ }**) and then the else will only trigger if **that** condition is true

- You can then have an else (with or without an if) after that etc. etc. If you want a catchall then that is just a final else with no if

Compound Conditionals

```
#include <iostream>

int main(){

  int test=4;
  int lowerbound = 1;
  int upperbound = 5;

 if (test>=lowerbound && test <= upperbound) {
    std::cout << "Test is within bounds\n";
  } else {
    std::cout <<
   “Test is less than or equal to candidate\n";
  }
}
```

- You can combine two tests with a logical operator
  - A && B means “if A **and** B are true”
  - A || B means “if A **or** B is true”
  - There are others, but that is the main one
- Logical operations are **short circuiting**
  - Working left to right, as soon as it is possible to know the truth of a compound statement the code stops operating
  - So for && this means that first false value stops and for || the first true value stops
  - This allows you to write something like
    - if (is_operation_valid && potentially_invalid_operation())
    - If the test for the operation being valid is false the operation never happens because of short circuiting

Functions

Functions

```
#include <iostream>

int demo_function(int i){
  return i+1;
}

int add_function(int i1, int i2){return i1+i2;}

void print_function(int i){std:cout << i << "\n";}

int main(){
  std::cout << demo_function(4) << "\n";
  std::cout << demo_function(7) << "\n";
  std::cout << add_function(1,2) << "\n";
  std::cout << print_function(add_function(demo_function(3), demo_function(4))) << "\n";
}
```

- C++ functions all have the same syntax
- **return_type function_name(parameter_one_type parameter_one_name, parameter_two_type ….)**
- If a function **doesn’t** return a value then there is a special keyword **void** to signify that

Functions

```
int demo_function(int i){
  return i+1;
}

int add_function(int i1, int i2){return i1+i2;}

void print_function(int i){std:cout << i << "\n";}
```

- demo_function takes one integer as a parameter and returns the parameter value + 1
- add_function takes two integers as parameters and returns their sum
- print_function takes on integer and prints it to screen, returning no value

Functions

```
int demo_function(int i){
  return i+1;
}
```

- One important thing to note about functions is that all of the parameters passed to a function are **copied** when they are passed
- So in **demo_function** the value of i inside the function **is** the value that you called the function with, but **changing** i inside the function doesn't have any effect outside the function
- That might be what you expect, but is not what some languages like Fortran do
- This is formally called **pass by value syntax**

References

References

- **References** are a very powerful tool in C++, but we're only going to talk about a fairly small part of them
- They are things that can be used like variables, but do not themselves hold a value
  - They **refer** to another variable that really does hold a value
- One of the many things you can do with references is allow you to pass parameters into functions without copying them
  - This can be very useful for passing large data objects into functions

Reference Parameters

```
#include <iostream>


int demo_function(int &i){
  i=i+1;
  return i;
}

int main(){
  int value=4;
  std::cout << demo_function(value) << "\n";
  std::cout <<
    "After calling demo_function, value is "
    << value << “\n";
}
```

- This is almost the same demo_function that we saw earlier
- I have just added an **&** to the parameter, after the type name but before the parameter name
- Inside the function I now modify the value of i by incrementing it and also return the incremented value
- You will note that I now call demo_function with a variable with the value of 4, not with the literal value of 4
- If I called it with 4, what does 4=4+1 mean?
  - You can actually do something almost like that with some nasty hacks in Java, but you can’t in most languages!

Reference Parameters

```
#include <iostream>

int demo_function(int &i){
  i=i+1;
  return i;
}

int main(){
  int value=4;
  std::cout << demo_function(value) << "\n";
  std::cout <<
    "After calling demo_function, value is "
    << value << “\n";
}
```

- This is almost the same demo_function that we saw earlier
- I have just added an **&** to the parameter, after the type name but before the parameter name
- Inside the function I now modify the value of i by incrementing it and also return the incremented value
- You will note that I now call demo_function with a variable with the value of 4, not with the literal value of 4
- If I called it with 4, what does 4=4+1 mean?
  - The compiler won’t let you compile code with a reference to a literal unless you also add **const** to the parameter to guarantee that you won’t change it

Vector and the Standard Library

Vector and the Standard Library

- One of the most powerful parts of C++ is the C++ standard library (often called the Standard Template Library or STL, but that is not formally correct)
- This is a library of common data structures and algorithms working on those data structures that is a part of the core language
- You as a programmer can always rely on them being there and don't have to write your own implementations of them or use a library
- We're going to discuss one of the simplest, but also one of the most powerful **std::vector**
- std::vector is basically an **array** in that it is a contiguous chunk of memory containing several of the same type of item one after the other accessed by a numerical index, but it can also grow when required

std::vector

```
#include<iostream>
#include<vector>//Vector has its own header

int main(){

  std::vector<int> vec;

  vec.push_back(1);
  vec.push_back(14);

 std::cout << "The first element is "
  << vec[0] << " and it should be 1\n”;
}
```

- Vector is a list of items (all of the same type, at least in normal code)
- You create an instance of a vector like any other variable, but you have to specify the type of the items that it contains in **< >** after the name **std::vector**
- Formally this is a **template parameter** but we aren’t going to discuss templates in this course

std::vector

```
#include<iostream>
#include<vector>//Vector has its own header

int main(){

  std::vector<int> vec;

  vec.push_back(1);
  vec.push_back(14);

 std::cout << "The first element is "
  << vec[0] << " and it should be 1\n”;
}
```

- Vector can contain an arbitrary number of items and you can always add new items to it using the **push_back** method
- This method adds your new item to the end of the vector
- When adding an item to a vector the item is copied, if you push_back a variable and then change the original variable it will not change the copy in the vector

std::vector

```
#include<iostream>
#include<vector>//Vector has its own header

int main(){

  std::vector<int> vec;

  vec.push_back(1);
  vec.push_back(14);

 std::cout << "The first element is "
  << vec[0] << " and it should be 1\n”;
}
```

- You access elements of a vector using the **[ ]** operator with the index of the item to get
- Vector indices run from **0** to **vec.size()-1**
- What is returned from the vector is a **reference** to the stored item so you can validly do **vec[0]=16** and the value in the vector will change

std::vector

```
#include<iostream>
#include<vector>//Vector has its own header

int main(){

  std::vector<int> vec;

  vec.push_back(1);
  vec.push_back(14);

 std::cout << "The first element is "
  << vec[0] << " and it should be 1\n”;

  vec.insert(vec.begin(),7);
}
```

- You can insert items into a vector using the **insert** method.
- The insert method requires you to specify where to insert into the vector using something called an **iterator**
- We aren’t going to get into iterators in detail, but **vec.begin()** gives you an iterator to the first element of the vector
- Hence, the code to the left inserts **7** before the first item in the vector which now reads **[7,1,14]**

std::vector

```
#include<iostream>
#include<vector>//Vector has its own header

int main(){

  std::vector<int> vec;

  vec.push_back(1);
  vec.push_back(14);

 std::cout << "The first element is "
  << vec[0] << " and it should be 1\n”;

  vec.insert(vec.begin(),7);
}
```

- If you want to insert somewhere else in the vector then you have to move the iterator
- The easiest way of doing this is by using the += operator
- **vec.begin()+=2** gives you an iterator to the third item in the vector

std::vector

```
#include<iostream>
#include<vector>//Vector has its own header

int main(){

  std::vector<int> vec;

  vec.push_back(1);
  vec.push_back(14);

 std::cout << "The first element is "
  << vec[0] << " and it should be 1\n”;

  vec.insert(vec.begin(),7);

  for (auto &value:vec){
    std::cout << value << " ";
  }
}
```

- The other really common thing to want to do is to loop over the elements in a vector
- There are several ways of doing this, including one using iterators (which is why they are called iterators!)
- The simplest one is shown to the left

std::vector

```
#include<iostream>
#include<vector>//Vector has its own header

int main(){

  std::vector<int> vec;

  vec.push_back(1);
  vec.push_back(14);

 std::cout << "The first element is "
  << vec[0] << " and it should be 1\n”;

  vec.insert(vec.begin(),7);

  for (auto &value:vec){
    std::cout << value << " ";
  }
}
```

- Here one instructs the compiler to loop over the elements in the **vec** container and get a reference to each element in the variable **value**

- Note the use of **auto** here because the compiler can easily work out that **value** should have the same type as an element of **vec**

std::vector

```
#include<iostream>
#include<vector>//Vector has its own header

int main(){

  std::vector<int> vec;

  vec.push_back(1);
  vec.push_back(14);

 std::cout << "The first element is "
  << vec[0] << " and it should be 1\n”;

  vec.insert(vec.begin(),7);

  for (auto &value:vec){
    std::cout << value << " ";
  }
}
```

- You don’t have to use a reference variable in this type of loop
- If you don’t then the value in the vector will be copied into the loop variable
- This does mean that you can’t accidentally change the value in the vector
- You usually find that you have missed that & when you want to change a value in a vector and find that it doesn’t update

Map

std::map

```
#include <iostream>
#include <map>
#include <string>

int main(){
  std::map<int,std::string> mymap;
  mymap[0]="Hello world";
  mymap[7]="Hello there";

  std::cout <<
   “Printing map element 7 – should be ""Hello there""\n";
  std::cout << mymap[7] << “\n";
}
```

- Here we map from the two values **0** and **7** to the two strings **Hello world** and **Hello there**
- While this looks a bit like a vector the elements 1-6 aren’t just empty, they don’t exist!
- Also, while the key here is an integer, it can be a very wide range of types

std::map

```
#include <iostream>
#include <map>
#include <string>

int main(){
  std::map<int,std::string> mymap;
  mymap[0]="Hello world";
  mymap[7]="Hello there";

  std::cout <<
   “Printing map element 7 – should be ""Hello there""\n";
  std::cout << mymap[7] << “\n";
}
```

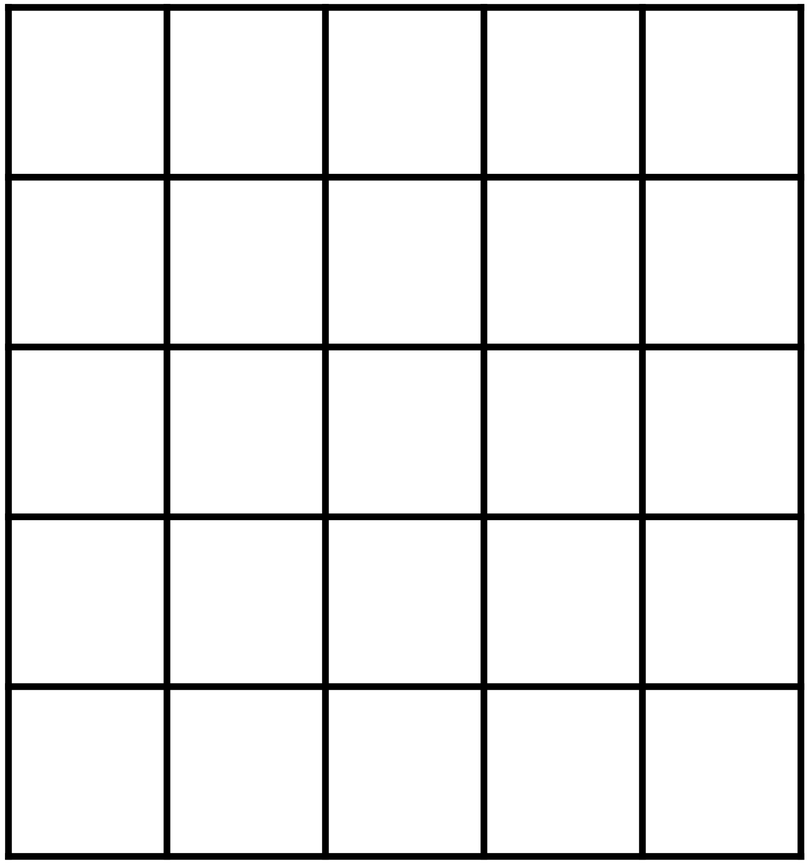
- There are few caveats for map, most notably that trying to access an element just silently creates the element it it didn’t already exist
  - You can see that that is basically how I initialised my two elements
- You can see more about map in the example pack

Multidimensional Arrays

Multidimensional Arrays

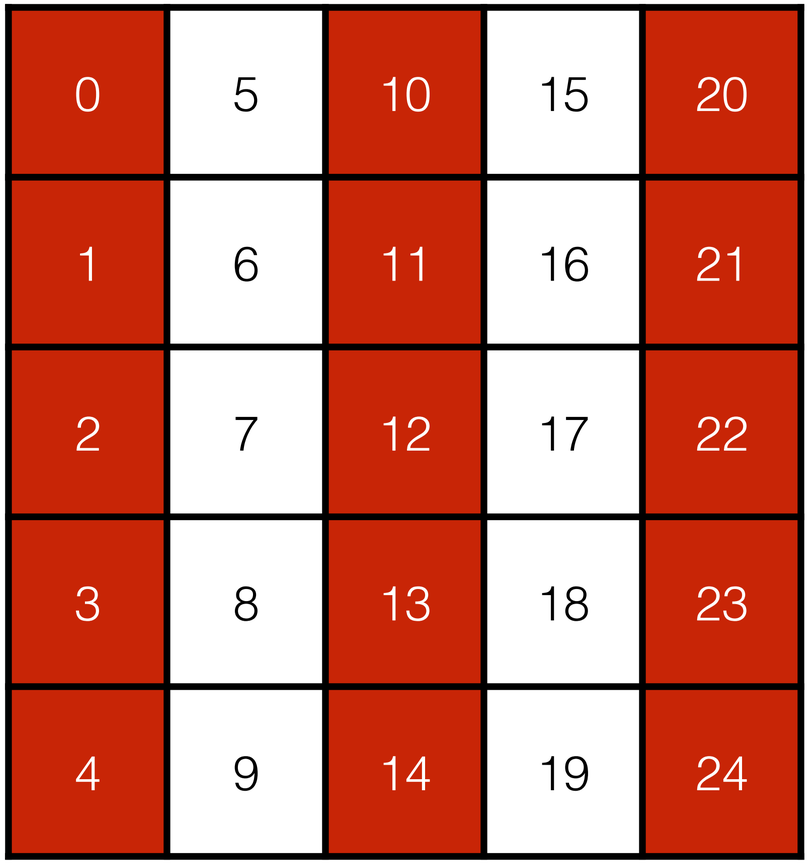
- If you want to have arrays that have more than one dimension (i.e. represent a 2D or 3D grid) **and** you want to be able to set the size at runtime there aren't really any really good options in C++
- The example pack has a few ways of doing it, ranging from using vectors of vectors (which technically isn't an array any more) to using a library called Boost to give you a true multidimensional array
- Here in the slides we'll describe a very simple way that works in any language - accessing a 1D array using an indexer function

Multidimensional Arrays



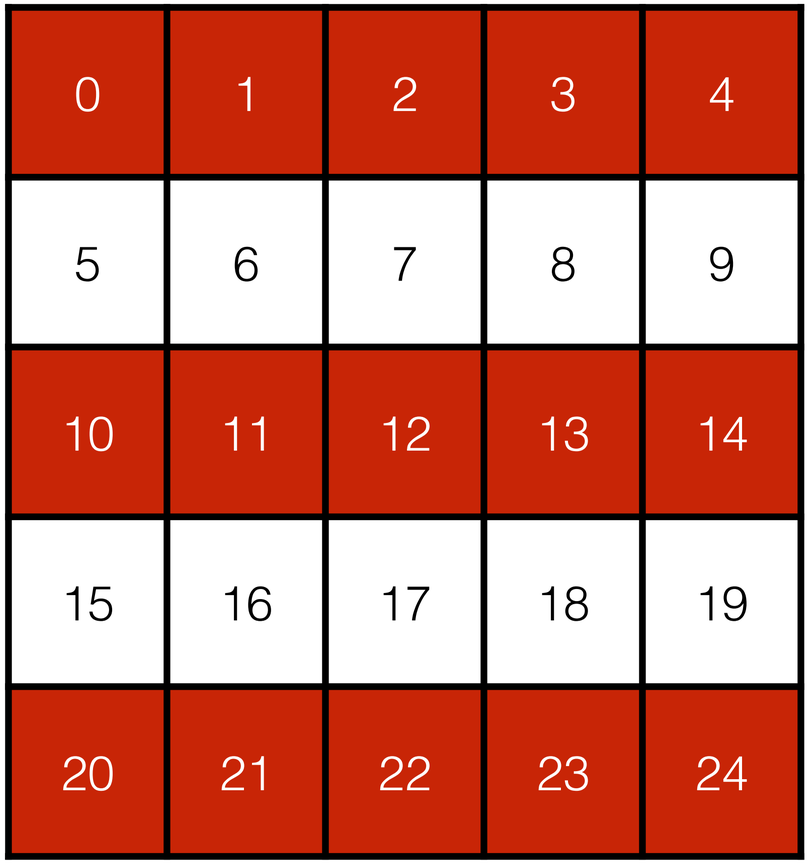
- Multidimensional arrays are not real in the computer hardware - the underlying memory can mostly be thought of as a 1D strip
- You **make** multdimensional arrays by ordering either the rows or the columns one after in the 1D strip

Multidimensional Arrays



- Multidimensional arrays are not real in the computer hardware - the underlying memory can mostly be thought of as a 1D strip
- You **make** multdimensional arrays by ordering either the rows or the columns one after in the 1D strip

Multidimensional Arrays



- Multidimensional arrays are not real in the computer hardware - the underlying memory can mostly be thought of as a 1D strip
- You **make** multdimensional arrays by ordering either the rows or the columns one after in the 1D strip

Multidimensional Arrays

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,2	2,3	2,4	2,5
3,0	3,1	3,2	3,3	3,4
4,0	4,1	4,2	4,3	4,4

- Either solution is as good at any other but generally C/C++ uses Row Major (second index fastest) and Fortran uses Column Major (first index fastest)
- So how do you actually use it? Well, you can see from the diagram
- Putting the coordinates of each cell on, you can see quite easily an element A_{ij} is at a location
  - $i*N_j + j$ (where here $N_j = 5$)

Multidimensional Arrays

```
#include <iostream>
#include <vector>


int main(){
  std::vector<int> arr_1dvector;
  arr_1dvector.resize(20*20);
  for(int i = 0; i < 20; ++i) {
    for(int j = 0; j < 20; ++j) {
     arr_1dvector[i*20+j] = i * j;
    }
  }
```

- Here is that converted to code
- You can see that I used the **resize** method of the vector to create 20*20 (400) elements without having to push them all back one by one
- Then I just access the elements as described
- Note that if I had done i+j*20 that array would be Fortran ordered