Scientific computing with C++

Richard F L Evans

Introduction

- Many programming languages - C, C++, Java, FORTRAN, C#, Go, Camel, Python, MATLAB...
- All have advantages and disadvantages - what is your objective?
  - Performance
  - Rapid prototyping
  - Portability
- Which to choose?

Some common choices of programming language

- Performance - FORTRAN, C, C++
- Rapid development - Python, MATLAB, R
- Portability - Java
- Which to choose?

Strengths of C++

- Compiled code - capable of high performance comparable with Fortran, C
- Flexible coding styles - Functional, object oriented, high level, low level
- Powerful standard library with many functions, more added with time (BOOST)
- Local scoping of variables (more later)
- Widespread adoption and support - cross platform, industry, academia

Disadvantages of C++

- A powerful and expansive tool - easy to code for coding's sake (over engineering)
- Matrices and arrays are horrible
- High performance code is harder to write (write for the compiler)
- Cryptic debugging for advanced features, and some not so advanced features

What about C?

- Isn’t C++ not just C with extra stuff?
- NOT the same language!
- Relies heavily on pointers to do things (pointers are evil, see later)
- Object orientation is ‘roll your own’ - bug prone and cumbersome
- A purely ‘low level’ language
- Archaic and no place in most software (only extremely performance and *memory* limited applications - not very common today)

KISS principle

- Keep It Simple and Stupid
- Very important for C++
  - Can write very elegant but impenetrable code in C++
  - Advanced features such as friend classes, inheritance, polymorphism, function pointers, templates, operator overloading increase complexity and make the code harder to understand and follow

C++

"A good FORTRAN programmer can write a good FORTRAN program in any programming language"

Hello World

Include files/libraries

iostream is part of the C++ standard library and allows input/output to screen

main() is a function of type int and is where all C++ programs start

```
#include <iostream>

int main(){

  std::cout << “hello world” << std::endl;

  return 0;

}
```

return statement to ‘return’ or end the program
(the function is of type int and so ‘0’ is returned, indicating success)

Hello World

```
#include <iostream>

int main(){

  std::cout << “hello world” << std::endl;

  return 0;

}
```

std is the ‘namespace’ for the standard library and contains a wide range of functions

cout is a ‘stream’ which prints variables and text to screen

defines the text to be printed to screen

special stream object which ‘ends the line’ and flushes the buffer (more later)

Hello World

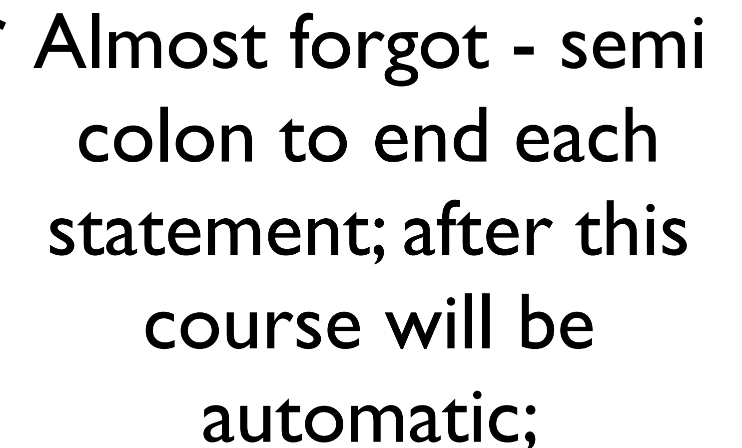
```
#include <iostream>

int main(){

   std::cout << “hello world” << std::endl;

   return 0;

}
```



Scope

Variable scope

- Defines where a variable is visible in a program
- Important and powerful concept
- Declare variables as you need them - not at the top of functions of the program

Simple example

```
#include <iostream>

int a=2; // visible everywhere -
         // a ‘global variable’

int main(){

  int b=5; // only visible in main()

  std::cout << a << “\t” << b << std::endl;

  return 0;

}
```

```cpp
#include <iostream>

int a=2; // visible everywhere -
         // a 'global variable' (bad)

int main(){

  int b=5; // only visible in main()

  // print out a+b 10 times
  for(int i=0; i<10; ++i){
     // declare c inside loop
     int c = a+b;
     std::cout << c << std::endl;
  }

  a = c; // error here - c is not visible
         // outside loop

  return 0;

}
```

Scoping with curly braces

```
#include <iostream>

int main(){

  {
    int b=5; // only visible here
  }
  std::cout << b << std::endl; // error

  return 0;

}
```

Namespaces

Namespaces

- A way to organize your code into logical modules
- Already seen one - the std namespace
- Can define your own and they serve the same purpose - to avoid naming conflicts
- Namespaces also logically divide your code and variables into discrete modules aka Good FORTRAN
- Alternative way to share variables between main and functions

Namespace syntax

```
namespace namespace_name
{

  // namespace variables

  // namespace functions

}
```

Namespace example

```
namespace car{
  // namespace variables
  int num_passengers;
  double position;
  double speed;
  // namespace functions
  double move_forward(){ return car::speed*10.0}
}

int main(){

    // set namespace variables
    car::position = 10.0;
    car::speed = 30.0;

    // use namespace function
    car::position += car::move_forward();

    return 0;
}
```

The C++ Standard Library

Standard Library

- A range of higher level functions and data structures to simplify code development
- Includes strings, mathematical functions, input and output, arrays, lists
- C++ is a minimal language - have to explicitly include library features using include statement:

```
#include <iostream>
```

Common functions

```
#include <iostream>    // Output to screen
#include <cmath>       // math functions
#include <vector>      // vector container
#include <string>      // text strings
#include <fstream>     // output to file
#include <sstream>     // output to string(!)
```

- More information as we go along
- Just remember that you need to include the right component for the part of the library you want to use

Allocatable Arrays

Allocatable array declaration

```
int main(){

  int * array; // declare a pointer

  array = new int[5000000]; // allocate array with 5000000 values

  array[1234]=34567; // assign value to array

  delete[] array; // deallocate memory

  return 0;

}
```

Allocatable array declaration

```
int main(){
  int * array; // declare a pointer
  array = new int[5000000]; // allocate array with 5000000 values
  array[1234]=34567; // assign value to array
  delete[] array; // deallocate memory
  return 0;
}
```

Do not use these

Allocatable arrays are bad

- Use pointers (works with raw memory addresses)
- Need to be allocated and then deallocated before exiting the program
- Undeallocated memory is a memory leak
- No checking of array bounds
- No way to find out the size of the array
- Messy to pass to other functions
- AND there is a much better way

Storage Containers

Several container types available in the standard library

- vector, map, list, deque, valarray, array
- Automatic memory management
- Know their own size
- Bounds checking available
- Neat built-in functions such as sorting, ordering (std::algorithm)

Vectors

```
#include <vector> // include vector header

int main(){

  std::vector<int> array(5); // array for storing five int variables

  array[3]=4; // once declared behaves just like a normal array

  return 0;

}
```

Vector declarations

```
#include <vector> // include vector header

int main(){

  std::vector<double> array1(10, 5.0); // array with 10 elements
                                       // all initialized to 5.0

  std::vector<double> array2; // empty array

  array2.resize(100000); // resize array2 to contain 100000 elements

  array2.resize(100, 5.0); // resize array2 to contain 100 elements
                           // each initialized to 5.0

  array1 = array2; // make a copy of array2 and save in array1

  return 0;

}
```

Vector functions

```
#include <vector> // include vector header

int main(){

  std::vector<double> array; // empty array

  array.reserve(1000);   // reserve storage for 1000 elements
  array.resize(100);     // resize array to contain 100 elements
  array.at(50) = 25.0;   // array access with bounds checking
  array.push_back(34.0); // increase array size by 1 and
                         // save the value 34.0

  // diagnostic functions
  unsigned int array_size = array.size(); // size of array
  unsigned int array_cap  = array.capacity(); // reserved array size

  return 0;

}
```

Using vectors

```
#include <vector> // include vector header

int main(){

  std::vector<float> array(20); // array of 20 floats

  // initialize values in array
  for(int i=0; i<array.size(); ++i){
    array.at(i) = 5.0f*float(i);
  }

  return 0;

}
```

- Very safe but slow way of accessing arrays as always check against size of array

Multidimensional vectors

```
#include <vector> // include vector header

int main(){
  // a vector of a vector of float - must have “> >”, not “>>”
  std::vector<std::vector <float> > array; // empty 2D array

  // set number of rows and columns
  int num_rows = 5;
  int num_cols = 10;
  array.resize(num_rows);
  for(int i=0; i<array.size(); ++i){
    array.at(i).resize(num_cols);
  }

  array[3][8] = 5.0f;          // fast
  array.at(2).at(9) = 10.0f; // bounds checking but slow

  return 0;
}
```

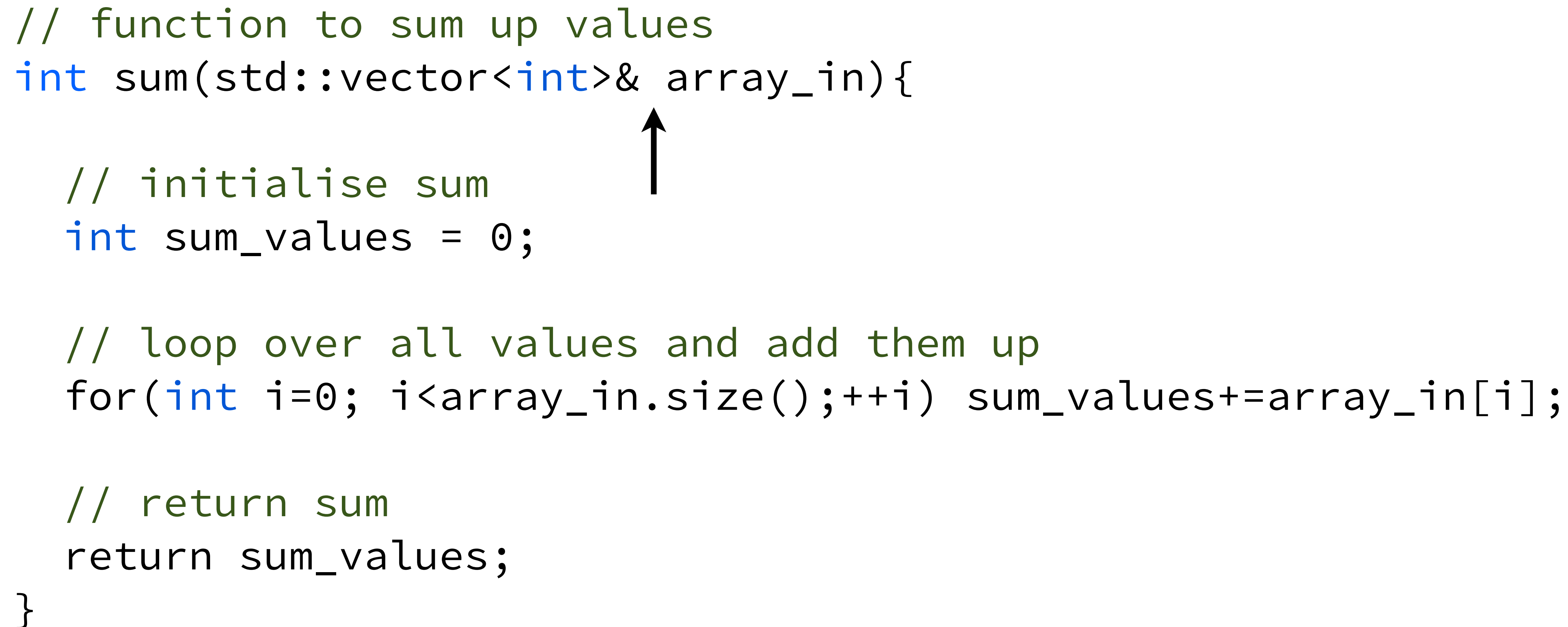
Passing vectors to functions

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

// function to sum up values
int sum(std::vector<int> array_in){
  // initialise sum
  int sum_values = 0;
  // loop over all values and add them up
  for(int i=0; i<array_in.size();++i) sum_values+=array_in[i];
  // return sum
  return sum_values;
}

int main(){
  std::vector<int> array(5,2.0);
  int sumv = sum(array); // call function and store result in sumv
  std::cout << sumv << std::endl;
  return 0;
}
```

Reference operator



- By default variables passed to functions are copied (very expensive for arrays)
- 'Reference' operator passes the actual variable(array) to the function (much faster)

Example functions using vectors

```
// function to zero values
void zero(std::vector<int>& array_in_out){

   // loop over all values and set to zero
  for(int i=0; i<array_in.size();++i) array_in_out[i] = 0;

  return; // note absence of variable

}

int main(){

  std::vector<int> array(5,2);

  zero(array); // zero array values

  return 0;
}
```

Example functions using vectors

```
// function returning vector<int>
std::vector<float> mul(std::vector<float> array_in, float a){

  // declare result array same size as array_in
  std::vector<float> result(array_in.size());

  // loop over all values and multiply by a
  for(int i=0; i<array_in.size();++i) result[i]=array_in[i]*a;

  return result;
}

int main(){

  std::vector<float> array(5,2.0f);

  array = mul(array, 5.0f); // multiply array by 5.0

}
```

list

```
#include <list> // include list header

int main(){

  std::list<int> mylist(5); // list of 5 int variables
  int count = 0;
// have to use iterators to access elements (set all elements to 78+c)
for(std::list<int>::iterator it=mylist.begin();it != mylist.end();++it){
  *it = 78+count;
  ++count;
}

  // a bit clunky but cool features
  mylist.sort(); // sort elements by number
                 // can even define custom sort function
  return 0;

}
```

list with vector

```
#include <algorithm> // cool functions for containers

int main(){
  std::vector<int> array;
  for(int i=0; i<100; ++i) array.push_back(i);  // set values in array

  std::list<int> mylist(array.size()); // list same size as array

  // copy to list
  copy(array.begin(), array.end(), mylist.begin());

  // sort elements by number
  mylist.sort();

  // copy back to vector
  copy(mylist.begin(), mylist.end(), array.begin());

  return 0;
}
```

list with vector

```
#include <algorithm> // cool functions for containers

int main(){
  std::vector<int> array;
  for(int i=0; i<100; ++i) array.push_back(i);  // set values in array

  std::list<int> mylist(array.size()); // list same size as array

  // copy to list
  copy(array.begin(), array.end(), mylist.begin());

  // sort elements by number
  mylist.sort();

  // copy back to vector
  copy(mylist.begin(), mylist.end(), array.begin());

  return 0;
}
```

std::algorithms that can be used with containers

all_of
Test condition on all elements in range (function template)
any_of
Test if any element in range fulfills condition (function template)
none_of
Test if no elements fulfill condition (function template)
for_each
Apply function to range (function template)
find
Find value in range (function template)
find_if
Find element in range (function template)
find_if_not
Find element in range (negative condition) (function template)
find_end
Find last subsequence in range (function template)
find_first_of
Find element from set in range (function template)
adjacent_find
Find equal adjacent elements in range (function template)
count
Count appearances of value in range (function template)
count_if
Return number of elements in range satisfying condition (function template)
mismatch
Return first position where two ranges differ (function template)
equal
Test whether the elements in two ranges are equal (function template)
is_permutation
Test whether range is permutation of another (function template)
search
Search range for subsequence (function template)
search_n
Search range for elements (function template)

Modifying sequence operations:
copy
Copy range of elements (function template)
copy_n
Copy elements (function template)
copy_if
Copy certain elements of range (function template)
copy_backward
Copy range of elements backward (function template)
move
Move range of elements (function template)
move_backward
Move range of elements backward (function template)
swap
Exchange values of two objects (function template)
swap_ranges
Exchange values of two ranges (function template)
iter_swap
Exchange values of objects pointed to by two iterators (function template)
transform
Transform range (function template)
replace
Replace value in range (function template)
replace_if
Replace values in range (function template)
replace_copy
Copy range replacing value (function template)
replace_copy_if
Copy range replacing value (function template)
fill
Fill range with value (function template)
fill_n
Fill sequence with value (function template)
generate
Generate values for range with function (function template)
generate_n
Generate values for sequence with function (function template)
remove
Remove value from range (function template)
remove_if
Remove elements from range (function template)
remove_copy
Copy range removing value (function template)
remove_copy_if
Copy range removing values (function template)
unique
Remove consecutive duplicates in range (function template)
unique_copy
Copy range removing duplicates (function template)
reverse
Reverse range (function template)
reverse_copy
Copy range reversed (function template)
rotate
Rotate left the elements in range (function template)
rotate_copy
Copy range rotated left (function template)
random_shuffle
Randomly rearrange elements in range (function template)
shuffle
Randomly rearrange elements in range using generator (function template)

Partitions:
is_partitioned
Test whether range is partitioned (function template)
partition
Partition range in two (function template)
stable_partition
Partition range in two - stable ordering (function template)
partition_copy
Partition range into two (function template)
partition_point
Get partition point (function template)

Sorting:
sort
Sort elements in range (function template)
stable_sort
Sort elements preserving order of equivalents (function template)
partial_sort
Partially sort elements in range (function template)
partial_sort_copy
Copy and partially sort range (function template)
is_sorted
Check whether range is sorted (function template)
is_sorted_until
Find first unsorted element in range (function template)
nth_element
Sort element in range (function template)

Binary search (operating on partitioned/sorted ranges):
lower_bound
Return iterator to lower bound (function template)
upper_bound
Return iterator to upper bound (function template)
equal_range
Get subrange of equal elements (function template)
binary_search
Test if value exists in sorted sequence (function template)

Merge (operating on sorted ranges):
merge
Merge sorted ranges (function template)
inplace_merge
Merge consecutive sorted ranges (function template)
includes
Test whether sorted range includes another sorted range (function template)
set_union
Union of two sorted ranges (function template)
set_intersection
Intersection of two sorted ranges (function template)
set_difference
Difference of two sorted ranges (function template)
set_symmetric_difference
Symmetric difference of two sorted ranges (function template)

Heap:
push_heap
Push element into heap range (function template)
pop_heap
Pop element from heap range (function template)
make_heap
Make heap from range (function template)
sort_heap
Sort elements of heap (function template)
is_heap
Test if range is heap (function template)
is_heap_until
Find first element not in heap order (function template)

Min/max:
min
Return the smallest (function template)
max
Return the largest (function template)
minmax
Return smallest and largest elements (function template)
min_element
Return smallest element in range (function template)
max_element
Return largest element in range (function template)
minmax_element
Return smallest and largest elements in range (function template)

Other:
lexicographical_compare
Lexicographical less-than comparison (function template)
next_permutation
Transform range to next permutation (function template)
prev_permutation
Transform range to previous permutation (function template)

valarray - designed for arrays of numerical values

```
#include <cmath>
#include <valarray>

int main(){

  // declare list of values
  double val[] = {9.0, 25.0, 100.0};

  // initialise valarray with values
  std::valarray<double> foo (val,3);

  // now square root all values and save in new valarray
  std::valarray<double> bar = sqrt (foo);

  return 0;

}
```

Struct - a user defined type

```
// define a new type car_t (_t is a good idea to indicate it’s a type)
struct car_t{
  int num_passengers;
  std::string color;
};

int main(){

    // define a variable of type car_t
    car_t red_car;

    // Set values in struct
    red_car.num_passengers = 2;
    red_car.color = “red”;

    // declare an array (vector) of cars
    std::vector<car_t> array_of_cars(10);
    array_of_cars[5].color = “green”; // set the color of the 6th car
    return 0;
}
```

Random numbers

New part of C++11 standard

- Implements a number of different and good generators with standard distributions
  - Linear congruential, Mersenne twister, Subtract-with-carry
- Distributions for each generator
  - Uniform, Bernoulli, Binomial, Geometric, Negative binomial, Poisson Exreme Value, Normal Lognormal Chi-squared, Cauchy Fisher-F Student-T, Discrete, Piecewise constant, Piecewise linear

RNG class example (C++)

```cpp
// simple wrapper class for rng
class rng{

   // std::random variables (internal to class)
   std::mt19937 mt; // mersenne twister
   std::uniform_real_distribution<double> dist;

public:

   // seed rng with uniform distribution [0:1)
   void seed(unsigned int random_seed){
      dist = std::uniform_real_distribution<double>(0.0,1.0);
      std::mt19937::result_type mt_seed = random_seed;
      mt.seed(mt_seed); // seed generator
   }

   // wrapper function generate a uniform random number between 0 and 1
   double grnd(){
      return dist(mt);
   }

};
```

Strings and IO

Standard library strings

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

int main(){

  std::string hello_text = “hello”;
  std::string world_text = “world”;
  std::string hello_world_text = hello_text + world_text;

  std::cout << hello_world_text << std::endl;

  return 0;

}
```

- A form of container, but just for characters
- Can be assigned, copied, concatenated (+)

Some useful characters

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

int main(){

  std::string tab = “\t”;
  std::string space = “ ”;
  std::string new_line = “\n”;
  std::string text = “hello world”;

  std::cout << text << tab << text << “\n” << std::endl;

  return 0;

}
```

File input and output

- Getting data into and out of your program is often necessary for storage of results, post processing, reading initial data etc
- In C++ this is done using 'streams' - analagous to text flowing down a stream

File input and output

```
#include <fstream> // header file for file i/o functions

int main(){

  std::ofstream ofile; // output file stream declaration
  std::ifstream ifile; // input file stream declaration

  // open files with a specified name
  ofile.open(“output_file_name”);
  ifile.open(“input_file_name”);

  // close the file
  ofile.close();
  ifile.close();

  return 0;

}
```

File output

```
#include <fstream> // header file for file i/o functions

int main(){

  int a=5;

  std::ofstream ofile; // output file stream declaration

  // open file
  ofile.open(“output_file_name”);

  // output some data to file
  ofile << “this is some text” << std::endl;
  ofile << a << std::endl;

  ofile.close();

  return 0;

}
```

High precision output

```
#include <fstream> // header file for file i/o functions
#include <iomanip> // functions for manipulating output formatting

int main(){

  std::ofstream ofile; // output file stream declaration
  ofile.open(“output_file_name”);
  double d = 1.23456;

  // output data with different precision
  ofile << std::setprecision(5) << d << std::endl; // 1.2346
  ofile << std::setprecision(8) << d << std::endl; // 1.23456
  ofile << std::fixed; // set fixed precision
  ofile << std::setprecision(8) << d << std::endl; // 1.2345600
  ofile.close();

  return 0;

}
```

Specify a filename at runtime

```
#include <fstream> // header file for file i/o functions
#include <sstream> // string streams

int main(){

  std::ofstream ofile; // output file stream declaration
  std::stringstream ss; // string stream declaration

  // construct file name
  ss << “output” << “file” << 123;

  // convert to string
  std::string ofile_name = ss.str();

  // cast as C-string when opening file
  ofile.open(ofile_name.c_str());

  ofile.close();
  return 0;
}
```

File input

```
#include <fstream> // header file for file i/o functions

int main(){

  int a;
  int b;

  std::ifstream ifile; // input file stream declaration

  // open file
  ifile.open("input_file_name");

  // read variables a and b from a file
  ifile >> a >> b;

  ifile.close();

  return 0;

}
```

File input reading whole lines

```
int a,b;

std::ifstream ifile(“input_file_name”);

std::string line; // declare a string to hold line of text

// Read in whole lines
getline(ifile,line);

// Convert line to stream
std::stringstream line_stream(line);

// Read in from line stream
line_stream >> a >> b;

ifile.close();
```

Fill arrays with data from file

```
std::vector<int> array_a,array_b;
std::ifstream ifile(“input_file_name”);
std::string line; // declare a string to hold line of text

while( getline(ifile,line) ){  // Read in all lines

  std::stringstream line_stream(line);  // Convert line to stream

  int a,b; // temporary variables

  // Read in from line stream
  line_stream >> a >> b;

  // add values to arrays
  array_a.push_back(a);
  array_b.push_back(b);
}

ifile.close();
```

Additional resources

- www.cplusplus.com/doc/tutorial
- http://www.parashift.com/c++-faq/index.html
- http://www.agner.org

Primitive types

FORTRAN	C++
`integer (2) [16-bit]`	`short int | int16_t [16-bit*]`
`integer | integer (4) [32-bit]`	`int | int32_t [32-bit*]`
`integer (8) [64-bit]`	`long int | int64_t [64-bit*]`
`real | real (4) [32-bit]`	`float [32-bit*]`
`double precision | real (8) [64 bit]`	`double [64 bit*]`
`character`	`char`
`logical`	`bool`

*C and C++ variable sizes are platform and compiler dependent, only specify a minimum

C++ operators

```
int b = 1;
int a = b; // assignment r-> l
a = a+1; // add one to a
a += 2; // add two to a
a++; // add one to a
a -= 1; // take one from a
a *= b; // multiply a*b and save the result in a
b = a/2; // divide a by 2 and save in b
== comparison
&& logical AND
|| logical OR
```