Crash Course in C++

R F L Evans

www-users.york.ac.uk/~rfle500/

Course overview

- Lecture 1 - Introduction to C++
- Lecture 2 - Functions and Data
- Lecture 3 - Namespaces and Files
- Lecture 4 - Code Organization
- Lecture 5 - Practical

Lecture 3
Namespaces and Files

- Standard string library
- File operations
- Input/output manipulation
- Namespaces
- Structs
- Coding style

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;

}
```

Can occasionally be problematic

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();
```

Binary Output

```
int a = 456;
std::vector<int> array(1000,123);

// open file in binary mode
std::ofstream ofile("output_file_name",std::ios::binary);

// directly write variables and arrays to file
ofile.write( reinterpret_cast<const char*>(&a),sizeof(int));
ofile.write( reinterpret_cast<const char*>(&array[0]),
             sizeof(int)*array.size() );
```

- Text data is flexible, but slow and imprecise for transferring data between programs
- Binary format useful for outputting large arrays

Binary Input

```
int a;
std::vector<int> array(1000);

// open file in binary mode
std::ifstream ifile("input_file_name",std::ios::binary);

// read a
ifile.read( (char*)&a,sizeof(int) );

// read array
ifile.read((char*)&array[0],sizeof(int)*1000);
```

- Similar functions for reading data

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
- 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;
}
```

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 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;
}
```

Coding style

Coding style

- Everyone develops a preferred style of coding
- Eg. variable names, whitespace, position of curly braces
- Consistent style aids readability

Some suggestions for style

- Use readable variable names, e.g. “number_of_cars” instead of “nc”
- Declare each variable on its own line and add a comment specifying what it is for
- Always indent nested code
- Try and keep functions small (<50 lines)
- Make use of whitespace to aid readability

Typical style

```
int main(){
    double a = 0.4, b = 3.3,r;
    r = sqrt(a*a + b*b);
    if(r < 1.0 && r > 2.0) std::cout << r << std::endl;
    return 0;
}
```

My style

```
//-------------------------------------------------------
// Program to calculate distance of point from origin
// (c) R F Evans 2014, Licensed under the FreeBSD License
//-------------------------------------------------------
int main(){

    double x = 0.4; // x-position
    double y = 3.3; // y-position

    // calculate distance from origin
    double radius = sqrt(x*x + y*y);

    // if r greater than 1 and less than 2 print result to screen
    if(r < 1.0 && r > 2.0){
      std::cout << radius << std::endl;
    }

    return 0;
}
```

Comments

- Intelligent comments help the reader (usually you) understand what the code does and why
- Every non-trivial statement should have a comment stating the intent of what you want the code to do

```
// if r greater than 1 and less than 2 print result to screen
if(r < 1.0 && r > 2.0){
  std::cout << radius << std::endl;
}
```

- Makes errors obvious - either error in logic or error in implementation

Practical time...