C++ for Numerical Programming - lectures 4-6

Martin Robinson

2018

Lecture 4 — pointers, functions, and references

Pointers

A variable's address in the computer's memory is called a pointer
If a variable has been declared by

```
int total_sum
```

then the address of total sum is given by `&total_sum`
`&total_sum` takes a constant value, because the address of total sum in the computer's memory was allocated when it was declared.
`&total_sum` is therefore known as a constant pointer

Variable pointers may be declared as follows

```
double* p_x;
int* p_i;
```

`p_x` is a variable pointer to a variable of type double, `p_i` is a variable pointer to an integer

Note that spacing can vary: `double* p_x` and `double *p_x` are equivalent, but the first is clearer.

If `p_x` and `p_y` are both to be declared as pointers this is done by

```
double *p_x, *p_y;
```

In the declaration

```
int *p_i, j;
```

`p_i` is a pointer to an integer, and `j` is an integer. It is generally clearer to declare each new pointer on a separate line.

The contents of the memory that a pointer p_x points to is given by *p_x, for example

```
double y, z;    // y, z store double precision numbers
double *p_x;    // p_x stores the address
                // of a double precision number
z = 3.0;
p_x = &z;       // p_x stores the address of z
y = *p_x + 1.0; // *p_x is the contents of the memory p_x,
                // i.e. the value z
```

Note here that *p_x means two different things, depending on where it is

Functions

The code on the next slide is an example containing a function that multiplies two double precision floating point numbers

Note the *function prototype* that is the second line of code

The function prototype tells the compiler about function's return value and parameters

The variable names `x` and `y` in the prototype are ignored by the compiler and don't have to be included. But including them can clarify the program

```cpp
#include <iostream>
double multiply(double x, double y); // function prototype

int main()
{
  double a = 1.0, b = 2.0, z;
  z = multiply(a, b);
  std::cout << a <<" times "<< b <<" equals "<< z <<"\n";
  return 0;
}

double multiply(double x, double y)
{
  return x * y;
}
```

A function may also return no value, and be declared as `void`

An example of a `void` function is shown on the next slide

The pass mark for an exam is 30 marks. This function prints out a message informing a candidate whether or not they have passed the exam

```cpp
#include <iostream>
void output(int score, int passMark);

int main()
{
  int score = 29, pass_mark = 30;
  output(score, pass_mark);
  return 0;
}

void output(int score, int passMark)
{
  if (score >= passMark)
    std::cout << "Pass - congratulations!\n";
  else
    std::cout << "Fail - better luck next time\n";
}
```

Any variables that are used in the function must be declared as in the main program

For example

```
float mult5(double x)
{
  float y = 5.0;
  return x * y;
}
```

Under most conditions a function can only change the value of a variable inside the function, and not in the main program

This is because the code makes a copy of the variable sent to a function, and sends this copy to the function

On return from the function changes in this copied variable have no effect on the original variable

For example the following function has no effect on the variable `x` outside the function

```
x = 2.0;
noeffect(x);
std::cout << x << "\n"; // will print out 2.0

void noeffect(double x)
{
  // x takes the value 2.0 here
  x += 1.0;
  // x takes the value 3.0 here
}
```

One method of allowing a function to change the value of a variable is to send the address of the variable to the function

```
#include <iostream>
void add(double x, double y, double* pz);

int main()
{
  double a = 1.0, b = 2.0, z;
  add(a, b, &z);
  std::cout << a <<" plus "<< b <<" equals "<< z <<"\n";
  return 0;
}

void add(double x, double y, double* pz)
{
  *pz = x + y;
}
```

On the previous slide, the variables `a` and `b` are sent to the function—these values cannot be changed by the function

We also send the address of `z` to the function - we therefore cannot change the address of `z`, but we can change the contents of `pz`

The contents of `pz` are changed in the function using the line of code

```
*pz = x + y;
```

References

Another way of allowing a function to change the value of a variable outside the function is to use *references*

These are much easier to use: all that has to be done is the inclusion of the symbol & before the variable name in the declaration of the function and the prototype.

For example, see the code on the next slide

```cpp
#include <iostream>
void add(double x, double y, double& rz);

int main()
{
  double x = 1.0, y = 2.0, z;
  add(x, y, z);
  std::cout << x <<" plus "<< y <<" equals "<< z <<"\n";
  return 0;
}

void add(double x, double y, double& rz)
{
  rz = x + y;
}
```

Lecture 5 — More on functions

Default values for function parameters

It is possible to allow a function to be called without specifying all the parameters needed

Default parameters will be used for the other parameters

These parameters should be declared in the function prototype

The arguments with default parameters must be the last parameters in the parameter list

This should be used with care: it is easy to forget that default parameters exist

For example, a solver may be written

```
void solver(float x, float epsilon, int maxiter)
{
  ...
}
```

The function prototype may be written

```
void solver(float x,
            float epsilon = 0.0001,
            int maxiter = 100);
```

This solver may be called using any of the following

```
solver(x, 0.01, 10000);
solver(x, 0.01); // default value used for maxiter
solver(x); // default value used for epsilon and maxiter
```

Function overloading

When a function is declared, the return type and parameter type must be specified

If a function `mult` is to be written that multiplies two numbers, we would like it to work for floating point numbers and for integers

This can be achieved by *function overloading*

More than one function `mult` can be written - one that takes two integers and returns an integer, one that takes two floating point numbers and returns a floating point number, etc

```
std::cout <<"7 times 10 equals "
          << mult(7, 10) <<"\n";
std::cout <<"21.5 times 14.5 equals "
          << mult(21.5, 14.5) <<"\n";

float mult(float x, float y)
{
  return x * y;
}

int mult(int x, int y)
{
  return x * y;
}
```

Function overloading also allows the definition of what is meant by multiplying a floating point number by an integer

Templated functions

- Templates introduce compile-time polymorphism: generics
- They are used where the same code may need to repeated for different values or for different types

```
double get_min(double a, double b)
{
   if (a<b){return a;} return b;
}

int get_min(int a, int b)
{
   if (a<b){return a;} return b;
}
```

Use the `template` keyword to produce as many functions as may be required

```
template <typename T>
T get_min (T a, T b) {
    if (a<b) {
        return a;
    }
    return b; // When a>=b
}

int main(void) {
   std::cout << get_min<int>(10,-2) << "\n";
   double ans = get_min<double>(22.0/7.0, M_PI);
}
```

Function template type inference

Note: it is not always necessary to provide the typename when calling the templated *function* (not class), as long as the compiler can infer it

```
int main(void) {
   int arg1 = 10;
   int arg2 = -1;
   std::cout << get_min(arg1,arg2) << std::endl;
}
```

Multiple template arguments

List multiple template arguments one after the other. These can be types (e.g. `typename T`) or non-types (e.g. `int N`)

```cpp
template <int N, typename T>
T multiply_by_n (T a) {
    return N*a;
}

int main(void) {
    int i = 1;
    std::cout << multiply_by_n<2>(i) << std::endl;
}
```

Lambda functions

You can define a function within the current scope using a *lambda* function

```
auto hello_world = []() {
      std::cout << "hello world" << std::endl;
   };
hello_world();
```

The `auto` keyword allows the compiler to determine the correct type for `hello_world`, rather than you defining it manually (very difficult for lambda functions!)

Lambda functions

The square brackets *capture* variables from the outside scope, for example

```
int i = 1;
auto add_i_to_arg = [i](int arg) { return arg + i; }
std::cout << add_i_to_arg(3) << std::endl; // prints 4
```

This captures `i` by value, to capture by reference use `&`

```
int i = 1;
auto add_arg_to_i = [&i](int arg) { i += arg; }
add_arg_to_i(3);
std::cout << i << std::endl; // prints 4
```

Lambda functions

You can capture all variables used in the lambda function using either [=], which captures everything by value, or [&], which captures everything by reference

```
int i = 1;
auto add_i_to_arg = [=](int arg) { return arg + i; }
std::cout << add_i_to_arg(3) << std::endl; // prints 4

auto add_arg_to_i = [&](int arg) { i += arg; }
add_arg_to_i(3);
std::cout << i << std::endl; // prints 4
```

Lecture 6 — Vectors

We have already covered a fixed-sized array, `std::array<T,N>`, but a much more flexible container is provided by `std::vector<T>`, which is a dynamically-sized vector holding elements of type `T`

```
std::vector<double> x = {1.0, 2.0, 3.0};
x.push_back(4.0);  // x now holds 4 values

std::vector<std::string> y;
y.push_back("one");
y.push_back("two");
```

Adding and removing elements

Add elements to the *end* of the vector using `push_back`, remove elements from the *end* of the vector using `pop_back`. You can resize the vector using `resize`. Get the current size of the vector using `size`

```
std::vector<double> x;
x.push_back(1.0);
x.push_back(2.0); // x holds {1.0, 2.0}
x.pop_back();     // x holds {1.0}
x.resize(3);      // x holds {1.0, ?, ?}

std::cout << x.size() << std::endl; // 3
```

Read/write individual values

You can access individual elements for reading or writing using `operator[]`

```
std::vector<int> x = {1, 2, 3};
std::cout << x[1] << std::endl; // print 2
x[1] = 5;
std::cout << x[1] << std::endl; // print 5
```

Read/write multiple values - loops revisited

You often want to loop through a vector to perform some operation, for example printing each element of a vector:

```
std::vector<double> x = {1, 2, 3, 4};
for (int i = 0; i < x.size(); ++i)
{
   std:cout << x[i] << std::endl;
}
```

You could call this an index-based for loop. Other methods of looping through a vector (or any other container in C++) include *range-based* loops, *iterator-based* loops, and the STL algorithms, for example. . .

Iterator-based loops

std::vector is a container in the Standard Template Library (STL). Every container defines its own *iterators*, which can be used to iterate through the container.

```
for (std::vector<double>::iterator i = x.begin();
     i != x.end(); ++i) {
   std:cout << *i << std::endl;
}
```

An iterator acts like a pointer to each element of the vector

Iterator-based loops using auto

The keyword auto is used to tell the compiler to compute the correct type (i.e. what is returned from x.begin(), this can be used to simplify this syntax somewhat. . .

```
for (auto i = x.begin(); i != x.end(); ++i)
{
   std:cout << *i << std::endl;
}
```

Range-based loops

Range-based loops have the most compact syntax, and work with any container that has `begin` and `end` methods.

```
std::vector<double> x = {1, 2, 3, 4};
for (double i: x)
{
   std:cout << i << std::endl;
}
```

Range-based loops using auto

You can use auto here to simplify things. . .

```
for (auto i: x)
{
   std:cout << i << std::endl;
}
```

Range-based loops using auto&

The code on the previous slide could not alter the contents of the vector because i was a *copy* of each element of x. You can instead make i a reference to edit values.

```
for (auto& i: x)
{
    i = 1.0; // set each element to 1.0
}
```

STL algorithms

The STL is vast, and we won't be covering even a small portion of it during this course, but there are many *algorithms* that operate on containers. For example, the `std::transform` algorithm

```
std::transform(x.begin(),x.end(),x.begin()
    [](double i) { return 2*i; });
```

You could also define the lambda function on its own as

```
auto f = [](double i) { return 2*i; };
std::tranform(x.begin(),x.end(),x.begin(),f)
```

Simply googling the name of the algorithm you require will lead you to the `cppreference.com` or `www.cplusplus.com` reference page fully describing how to use it.

A matrix using std::vector<double>

To create a two-dimensional array of double precision numbers with 5 rows and 3 columns called A we use the following section of code

```
std::vector<std::vector<double>> A;
A.resize(5);
for (int i=0; i<5; i++)
{
    A[i].resize(3);
}
```

A lower triangular matrix

Suppose we want to define a lower triangular matrix `A` of integers with 10,000 rows and 10,000 columns

```
std::vector<std::vector<double>> A;
A.resize(10000);
for (int i=0; i<10000; i++)
{
    A[i].resize(i+1);
}
```

Tip — Including Third-party Libraries

One of the advantages to using C++ is the wide availability of high performance libraries for scientific computing

Assuming you have installed a third-party library on your system, you can use it by including and linking against the relevant files

The easiest libraries to use are header-only libraries, such as the linear algebra library Eigen (see example on next slide). The `-I` flag for g++ specifies the directory where the header files (`.h`) are installed. These are the files that you include in your code using `#include`.

```
g++ -I/usr/include program.cpp
```

Example: Eigen (eigen.tuxfamily.org)

C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms

It is a header-only library, so there is no need to link any files. A simple Matrix example:

```
#include <iostream>
#include <Eigen/Dense>

using Eigen::MatrixXd;
int main()
{
  MatrixXd m(2,2);
  m(0,0) = 3;
  m(1,0) = 2.5;
  m(0,1) = -1;
  m(1,1) = m(1,0) + m(0,1);
  std::cout << m << std::endl;
}
```

Example `Makefile` for Eigen

```
EIGEN_INCLUDE = /usr/include/eigen3/

all: printMat

printMat: printMat.cpp
  g++ -I$(EIGEN_INCLUDE) -o printMat printMat.cpp
```