Introduction to C++ - lectures 3-4

Martin Robinson

2019

Lecture 3

- pointers
- functions
- references

Pointers

A pointer that points to an object **represents the address** of the first byte in memory occupied by the object.

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 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.

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 z = 3.0;
double* p_z = &z;       // store the address of z

double y = *p_z + 1.0; // *p_z is the contents of the memory that
                       // p_z points to, i.e., z
```

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

`[< compiler explorer >]`

Functions

The following **function prototypes** declare two functions: one called `my_func` that takes two parameters of type `double` and returns a variable of type `double`, and one called `main` that takes no parameters and returns an `int`:

```
double my_func(double x, double y);
int main();
```

The function prototype tells the compiler about function's name, return type, and parameters.

```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) // function definition
{
  return x * y;
}
```

`[< compiler explorer >]`

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 normal.

For example:

```
double multiply_by_5(double x)
{
  double y = 5.0;
  return x * y;
}
```

A function can only change the value of a variable inside the function, and not in the main program.

This is because, by default, variables are **passed by value**, and the function only sees a **copy**.

Changes in this copied variable have no effect on the original variable:

```cpp
#include <iostream>

void no_effect(double x) {
  x += 1.0;
}

int main() {
  double x = 2.0;
  no_effect(x);
  std::cout << x << '\n';
}
```

[< compiler explorer >]

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) {
  *pz = x + y;
}

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

On the previous slide, the variables `x` and `y` are copies of the variables `a` and `b`. The original `a` and `b` cannot be changed by the function.

But, we also send the address of `z` to the function and we **can** change the value that `pz` points to.

The contents of `pz` are changed in the function by 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;
}
```

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.

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

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

int main() {
  int i = mult(7, 10);
  float f = mult(21.5f, 14.5f);
}
```

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 Number>
Number get_min (Number a, Number b) {
    if (a < b) {
        return a;
    }
    return b;
}

int main(void) {
   int i = get_min<int>(10,-2);
   double d1 = get_min<double>(22.0/7.0, 3.14159265359);
   double d2 = get_min(22.0/7.0, 3.14159265359);
}
```

[< compiler explorer >]

Function template type deduction

Note: it is not always necessary to provide the typename when calling a templated function, 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

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

```
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 **lambda function** within the current scope:

```
auto empty_lambda = [](){};

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

hello_world();
```

The `auto` keyword allows the compiler to determine the correct type for the lambda, rather than you declaring it manually (impossible 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
```

Vectors - dynamic arrays

We have already covered a fixed-sized array, `std::array<T, N>`, but a 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 = {1.0, 2.0, 3.0, 4.0};
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 standard library algorithms:

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 over 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 infer the correct type (i.e. what is returned from `x.begin()`, this can be used to simplify this syntax:

```
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 = {1.0, 2.0, 3.0, 4.0};
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
}

for (const auto& i: x)
{
    std::cout << i << std::endl; // print each element to the console
}
```

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)
```

`[< cpp reference >]`

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 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(10'000);
for (int i=0; i<10'000; 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 <Eigen/Dense>
int main() {
  Eigen::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);
  auto m2 = m * m;
}
```

Example `Makefile` for Eigen

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

all: printMat

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