School of Computing
FACULTY OF ENGINEERING

Introduction to C++ Programming

Day 2: Beyond The Basics

Nick Efford

Email: N.D.Efford@leeds.ac.uk
Twitter: @python33r
Google+: http://gplus.to/pythoneer

Main Objectives For Today

- To explore in more detail how we can store collections of values and manipulate those collections
- To consider a feature of C++ that allows better structure in larger programs and makes code easier to reuse

Today's Topics

- STL containers and algorithms
- Arrays and pointers
- File I/O in C++
- Defining and using functions

The Standard Template Library (STL)

Containers

- `vector`, for storing a sequence of values
- ...plus other types, with different characteristics

Iterators

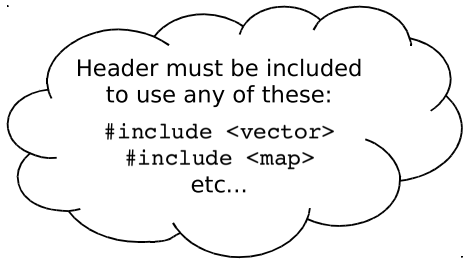
- Standard mechanism for moving through a container

Algorithms

- Standard tools for searching, sorting, transforming collections of values in various ways...

STL Containers

- Two types: sequences and associative containers
- Sequences preserve the order of insertion of items
  - Examples: `vector`, `list`, `deque`
- Associative containers maintain a sorted order for their contents – order of insertion is not preserved!
  - Examples: `set`, `map`



Vector Reminder

- Create in various ways:

```
vector<int> a;          // empty vector
vector<double> b(10);   // 10 elements, all 0
```

- Add values with `push_back` method:

```
vec.push_back(42);
```

- Determine size with `size` method:

```
vec.size()
```

- Access elements with `[]` or `at` method:

```
vec[0]      // accesses first element
vec.at(0)
```

Now do Exercise 1...

Lists Are Similar...

```
list<int> a;                // empty list of integers
list<int> b(10);            // 10 integers, all 0
list<int> c(5, -1);         // 5 integers, all -1
list<int> d(c);             // copy of c

list<string> p;             // empty list of strings
list<string> q(5, "xyz");   // 5 strings, all "xyz"
list<string> r(q);          // copy of q

p.push_back("Hello");       // adds string to end of p
cout << p.size() << endl;   // prints size of p
cout << p[0] << endl;       // COMPILER ERROR
```

Now do Exercise 2...

Methods Common to Vectors & Lists

`empty`	Returns `true` if container is empty
`size`	Returns number of items in container
`clear`	Empties container of all stored items
`push_back`	Adds item to back of container
`pop_back`	Removes item from back of container
`insert`	Inserts item at a specified position
`erase`	Remove item(s) at specified position(s)

Why Do We Need Lists?

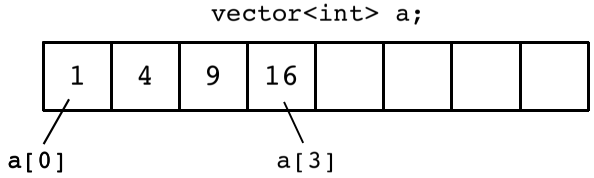
`vector`

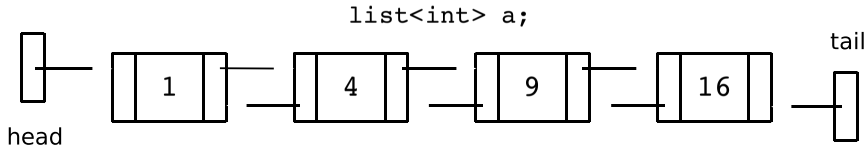
- Contiguous sequence of storage locations
- Good for random access to arbitrary items of data
- Poor performance when inserting/removing – especially at the front

`list`

- Doubly-linked sequence of nodes, each containing an item of data and two pointers
- Random access no longer possible
- Insertion/removal anywhere in list is efficient

Vectors vs. Lists





Methods Exclusive to Lists

`push_front`	Adds an item of data to front of list
`pop_front`	Removes an item from front of list
`unique`	Eliminates adjacent duplicate items
`reverse`	Reverses ordering of list
`sort`	Sorts items into order

vectors can be sorted using STL's algorithms library

Moving Through a Vector

```
vector<double> v;

...

for (int i = 0; i < v.size(); ++i) {
  cout << v[i] << '\n';
}
```

Same approach won't work for lists, so we need a more general technique...

Iterators

- ‘Pointers’ to a value stored in a container
- Can be incremented or decremented to move forward or backward through the container
- Can be ‘dereferenced’ to access container values
- Containers have `begin` and `end` methods, returning iterators that span the container

Examples of Iterator Types

```
list<string>::iterator
list<string>::const_iterator
```

can’t be used to change container

```
list<string>::reverse_iterator
list<string>::const_reverse_iterator
```

Useful abbreviation technique:

```
typedef list<string>::const_iterator Iterator;
```


Examples of Iterator Use

```
vector<double> v;
...
vector<double>::const_iterator i;
for (i = v.begin(); i != v.end(); ++i) {
  cout << *i << '\n';
}
```

```
vector<double> v;
...
vector<double>::iterator i;
for (i = v.begin(); i != v.end(); ++i) {
  *i *= 0.5;
}
```


STL Algorithms

- Functions that manipulate containers, arrays & strings in various ways using iterators
- Mostly accessed using `#include <algorithm>`
- Over one hundred available
  - Initialisation (e.g., `copy`)
  - Sorting & searching (e.g., `sort`, `find`)
  - In-place transformations (e.g., `reverse`, `replace`)
  - Scalar calculation (e.g., `count`, `count_if`)
  - Sequence generation (e.g., `generate`, `transform`)
  - Set operations (e.g., `merge`)

Example:
Sorting a Vector of Numbers

Ascending order

```
vector<int> data;
...
sort(data.begin(), data.end());
```

Descending order

```
vector<int> data;
...
sort(data.begin(), data.end(), greater<int>());
```

Now do Exercise 3...

Example: Removing Zeroes
From a List of Numbers

First occurrence

```
list<int> data;
...
list<int>::iterator i;
i = find(data.begin(), data.end(), 0);
if (i != data.end())
  data.erase(i);
```

All occurrences

```
list<int> data;
...
list<int>::iterator i;
i = remove(data.begin(), data.end(), 0);
data.erase(i, data.end());
```


Other Examples

```
vector<int> v;
...

// reverse ordering of elements
reverse(v.begin(), v.end());

// count number of zeros in v
int n = count(v.begin(), v.end(), 0);

// sum the values in v
int sum = accumulate(v.begin(), v.end(), 0);
```

#include <numeric> needed to use this

initial value, to which vector's values will be added

Representing 2D Matrices
Using Vectors

- Think of matrix as a 'vector of vectors'
  - Each row is a vector of numeric values
  - Matrix is a vector of these rows
- Bit fiddly – generally better to use a dedicated **matrix class** designed for such work

```
typedef vector<double> row;
typedef vector<row> matrix2d;
...
matrix2d a(4, row(5));       // 4x5 matrix of zeroes
matrix2d b(5, row(3, 1));    // 5x3 matrix of ones
cout << a[0][0];             // element at row 1, col 1
cout << a[3][4];             // element at row 4, col 5
```

Lower-Level Storage: Arrays

Definition Syntax

type array-name[*array-size*];
type array-name[*array-size*] = {*val1, val2*...};

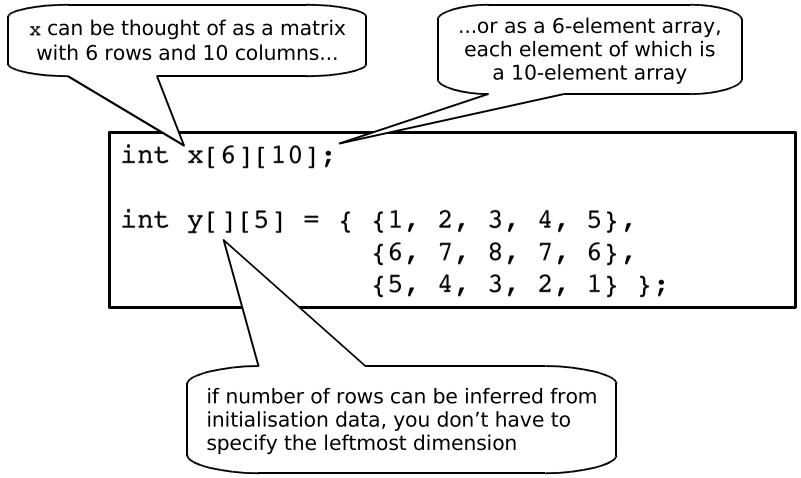
Examples

```
double x[3];
int y[5] = {10, 15, 20, 25, 30};

x[0] = 1.5;
cout << y[4] << endl;
```

- Indices of *N*-element array run from 0 to *N*–1, as for vectors
- No run-time error checking on array bounds
- No initialisation of array elements to a default value
- **Array size fixed at compile time!**

Multidimensional Arrays

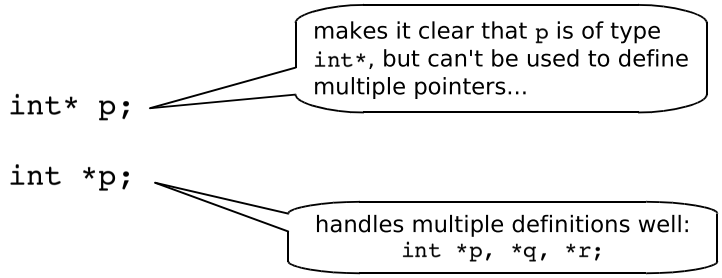


Pointers

- A C++ variable represents a fixed location in memory, into which a value (or values, in the case of arrays) can be stored
- Sometimes it is useful to work with **pointers** to memory locations, rather than the memory locations directly
- Pointer variables are essentially memory addresses that can be manipulated in various ways
- Moving a pointer to data around a program can be more efficient than moving the data around (less copying)
- Pointers allow us to allocate storage dynamically...

Pointer Definition

Pointer variables are defined using `*`:



Use whichever style you prefer...

Pointer Addressing & Dereferencing

```
int x = 42;
int* ptr = &x;

cout << x << endl;

cout << ptr << endl;

cout << *ptr << endl;


x = -1;
cout << *ptr << endl;

*ptr = 7;
cout << x << endl;
```

`&` is 'address of' operator; `ptr` now points at memory used by `x`

prints address of `x`

dereferences pointer; prints `42`

prints `-1`

prints `7`

Quiz

```
1  int *p, *q;
2  int x = 100, *r = &x;
3  q = r;
4  *r = -5;
5  *q = 256;
6  *p = 42;
```

1. What does line 3 do?
2. After line 5 executes, what is the value of `x`?
3. Is line 6 OK? If not, why not?

Answers

Dynamic Storage: `new` & `delete`

Pointers are useful for allocating storage **dynamically**:

```
cout << "How many numbers?" << endl;
cin >> size;
int* data = new int[size];
```

unlike arrays, size is determined **at run time**

...but we **must remember to release the storage**:

```
delete [] data;
```

Vectors vs. Arrays and Dynamic Storage

- Vectors are generally the easiest to use
  - Can grow and shrink
  - Useful methods
  - Bounds checking with `at`
- Arrays are a little more efficient and might be a more convenient choice if size will never change
- Dynamic storage gives you complete control but you need to write the memory allocation code yourself and remember to release storage with `delete`

File I/O in C++

- You need to `#include <fstream>`
- Use `ifstream` object to read from a file, `ofstream` object to write to a file, `fstream` object to do both
- If dealing with text files
  - Use >> to read values, as with `cin`
  - Use << to write values, as with `cout`
- If dealing with binary files
  - Use `read` method to read bytes
  - Use `write` merthod to write bytes

Example: Reading Numbers From a Text File

```
ifstream infile("numbers.txt");

if (! infile) {
  cerr << "Can't open file!" << endl;
  exit(1);
}

double value;
while (infile >> value) {
  ...
}
```

need `#include <cstdlib>` to use this; if in `main`, you can use `return 1;` instead

Now do Exercise 4...

Example: Writing a Vector of Numbers to a Text File

```
ofstream outfile("output.txt");

if (! outfile) {
  cerr << "Can't open file!" << endl;
  exit(1);
}

vector<double>::const_iterator i;
for (i = data.begin(); i != data.end(); ++i) {
  outfile << *i << '\n';
}

outfile.close();
```

ensures that buffered data are written to disk

Defining Functions

- Basic syntax
- Parameters & arguments

Function Definition Syntax

[*storage-class*] *return-type name* (*param-list*)
{
  [*variable-definitions*]
  [*statements*]
}

- *name* consists of letters, digits and underscore
- *param-list* is `void` if there are no parameters
- *return-type* can be `void`, or any type except an array
- If *return-type* isn't `void`, *statements* must include a `return` statement

Examples

One function

```
#include <iostream>

int main()
{
  std::cout << "Greetings!" << std::endl;
  return 0;
}
```


Examples

Two functions

```
#include <iostream>

void greet()
{
  std::cout << "Greetings!" << std::endl;
}

int main()
{
  greet();
  return 0;
}
```

Examples

Is this OK?

```
#include <iostream>

int main()
{
  greet();
  return 0;
}

void greet()
{
  std::cout << "Greetings!" << std::endl;
}
```


The Function Prototype

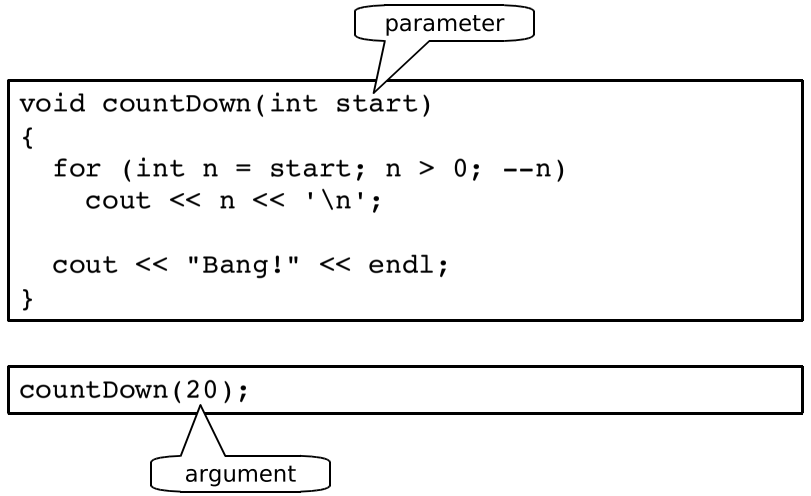
```
void greet();
double sqrt(double);
void readFile(const string&, int[], int);
```

- Tells the programmer what she needs to know to call a function correctly
- Also tells the compiler what it needs to know to generate code that calls the function
- Usually appears in **header files**

Parameters & Arguments

- Data are passed into a function via its **parameter list**, a comma-separated list of **formal parameters**
- Each formal parameter has a type and a name
- Functions must be called with a list of **arguments** that matches the parameter list
- Within the function body, parameters represent the arguments specified in a function call

Example



Default Arguments

used if no argument supplied by caller

```
void countDown(int start = 10)
{
  for (int n = start; n > 0; --n)
    cout << n << '\n';

  cout << "Bang!" << endl;
}
```

```
countDown();
countDown(20);
```


How Can a Function Alter Its Arguments?

```
void swap(int a, int b)
{
  int tmp = a;
  a = b;
  b = tmp;
}

int main()
{
  int x = 5, y = 2;
  swap(x, y);
  cout << x << ',' << y << endl;
  return 0;
}
```

What does this program print?

References

Like pointers, except

- Must point to some storage
- Can't be made to point anywhere else
- No special dereferencing syntax

Think of a reference as an **alias** for another variable:

```
int& x = value;
```

If a parameter is a reference, then it refers to the variable used as an argument – so changes made to the parameter will change that variable!

A Working Swap Function Using References

```
void swap(int& a, int& b)
{
  int tmp = a;
  a = b;
  b = tmp;
}

int main()
{
  int x = 5, y = 2;
  swap(x, y);
  cout << x << ',' << y << endl;
  return 0;
}
```

Older Pointer-Based Approach (Inherited From C)

```
void swap(int* a, int* b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main()
{
  int x = 5, y = 2;
  swap(&x, &y);
  cout << x << ',' << y << endl;
  return 0;
}
```

note how function arguments are addresses of variables that need to be changed

Don't do this in C++!

Function Overloading

```
void swap(int& a, int& b)
{
  ...
}

void swap(double& a, double& b)
{
  ...
}
```

same name, different parameter types

Overloading gives the appearance of a single function, but programmer still has to write both!

Passing Containers to Functions Efficiently

Pass a reference if the function modifies the container:

```
void readData(vector<double>& data)
{
  ...
}
```

Pass a `const` reference if it doesn't:

```
double mean(const vector<double>& data)
{
  ...
}
```


Summary

We have

- Explored vector in more detail and compared it with other containers such as list
- Considered how iterators and algorithms are used with C++ container types
- Discussed lower-level storage using fixed arrays or pointers + new & delete
- Examined the steps involved in file I/O
- Looked at how programs can be made more modular by writing functions