School of Computing
FACULTY OF ENGINEERING

Introduction to C++ Programming

Day 3: Libraries, Classes & OOP

Nick Efford

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

Main Objectives For Today

- To explore issues relating to the structure and compilation of larger programs
- To consider how we can make reusing code easier
- To introduce the idea of object-oriented programming

Course feedback: http://www.survey.leeds.ac.uk/itfeedback

Today’s Topics

- Command line arguments
- Multi-file compilation
- Makefiles
- Creating and linking with libraries
- Basic features of C++ classes

The Command Line

- What you type when running a program from a Linux terminal window or Windows Command prompt
- Always starts with name of program
- May be followed by **command line arguments** (CLAs)
- CLAs can be accessed within your program and therefore provide a way of feeding input to it

```
$ ./prog input.data output.data
```

CLAs in C & C++

- Alter `main` so it takes two parameters – an `int` value and an array of pointers to `char`:

```
int main(int argc, char* argv[])
{
  ...
}
```

- `argc` is the **argument count**: the number of things typed on the command line (including program name)
- `argv` is the array of arguments
  - `argv[0]` is program name
  - `argv[1]` is first thing typed after program name
  - `argc-1` is last valid index into `argv`

Modular Programming

- Programs can be structured as collections of **functions**, one of which (`main`) provides the entry point
- This makes programs easier to write and easier for others (or yourself in the future) to read and understand
- A big program might consist of a great many functions, and there are advantages to distributing these functions across a number of files...

Example: `stats.cpp`

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

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

double median(vector<double> data)
{
  ...
}

int main(int argc, char* argv[])
{
  ...
}
```

Alternative Physical Representation

```
void readData(istream& input, vector<double>& data)
{
  ...
}                                                         data.cpp
```

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

double median(vector<double> data)
{
  ...
}                                                         calc.cpp
```

```
int main(int argc, char* argv[])
{
  ...
}                                                        stats.cpp
```

Why Should We Split a Program Into Multiple Files?

- Efficiency
  - Reduced time to edit
  - Reduced time to recompile
- Teamwork
  - People can work on different files simultaneously
- Reuse

Naïve Approach

```
g++ file1.cpp file2.cpp file3.cpp -o prog
```

<u>All</u> source files are compiled...

...even if `file2.cpp` is the
only one that has changed!

Better Approach

```
g++ -c file1.cpp
g++ -c file2.cpp
g++ -c file3.cpp
g++ file1.o file2.o file3.o -o prog
```

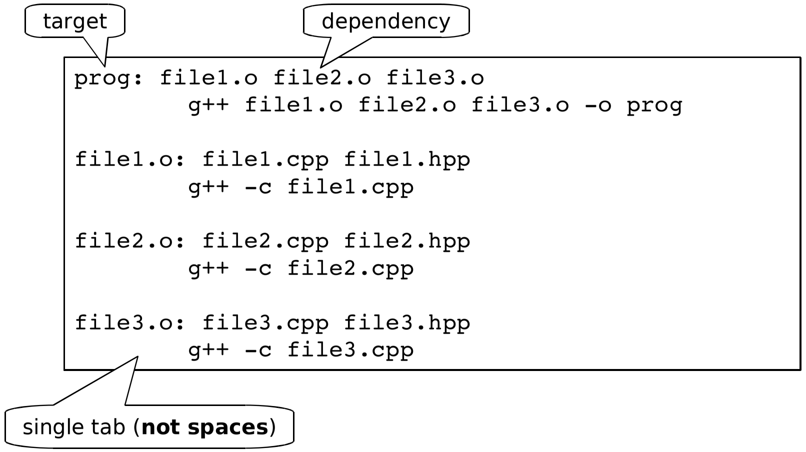
Source files compiled to object code separately

Compilation is separate from linking

If one file changes, we <u>recompile that file only</u> and relink

Now do Exercise 2...

Managing The Process: Makefiles



```
prog: file1.o file2.o file3.o
        g++ file1.o file2.o file3.o -o prog

file1.o: file1.cpp file1.hpp
        g++ -c file1.cpp

file2.o: file2.cpp file2.hpp
        g++ -c file2.cpp

file3.o: file3.cpp file3.hpp
        g++ -c file3.cpp
```

Now do Exercise 3...

Creating Static Libraries (UNIX-like Systems)

1. Put function prototypes, etc, in header files
2. Compile source files into object code
3. Bundle `.o` files into a single **archive**, using *ar*
4. Run *ranlib* to generate an 'archive index'

```
g++ -c file1.cpp
g++ -c file2.cpp
ar -crv libfunc.a file1.o file2.o
ranlib libfunc.a
```

Using Libraries

- Install header files and library archive in appropriate directories
- Use `-I` compiler option to specify the directory that the preprocessor should search for header files
- Use `-L` option to tell linker which directory to search for library archives
- Use `-l` option to tell linker which archive to use

```
g++ -c -I/usr/local/include prog.cpp
g++ prog.o -o prog -L/usr/local/lib -lfunc
```

Now do Exercise 4...

Shared Objects (DLLs)

Static linking

- Copies object code from library archives into an executable file
- Executables can become very large on disk
- Wastes memory on a multitasking OS: do we *really* need 30 copies of a function’s object code in memory at the same time?...

Dynamic linking

- Allows one copy of library code to exist in memory at runtime, referenced dynamically by multiple programs

Using Multiple Libraries

Imagine that we have two third-party libraries, for which no source code is available, each containing a function that we wish to use in our program...

What if the two functions perform similar tasks, and have therefore been given the same name by their respective authors?

Namespaces

Classes, objects and functions of the C++ standard library are defined in a **namespace** called `std`, which is used to prefix their names

We are free to use `cout` as a variable, because it won’t clash with `std::cout`

If we know there will be no clashes, we can import names from a namespace and avoid having to use the prefix:

```
using namespace std;
```

Namespace Definitions

```
namespace temperature
{
  double fahrenheit(double c)
  {
    return 9*c/5 + 32;
  }

  double celsius(double f)
  {
    return 5*(f-32) / 9;
  }
}
```

function’s full name is `temperature::celsius`

Classes & Object-Oriented Programming

- What are objects?
- What are classes?
- What are C++ classes?

Objects in the World

The world consists of **objects** with distinct **identities**

- Physical objects: this room, me, you...
- Conceptual objects: this course, your bank account...

Objects have **state**, captured by **attributes**

- This room: name, capacity...
- Your bank account: account number, balance...

Objects have **behaviour**

- This room: turn lights off, activate projector...
- Your bank account: deposit, withdraw, check balance...

Object Example

What attributes does this object have?

What are their values?

How does this object behave?

Classes

Objects can be assigned to different categories or **classes**

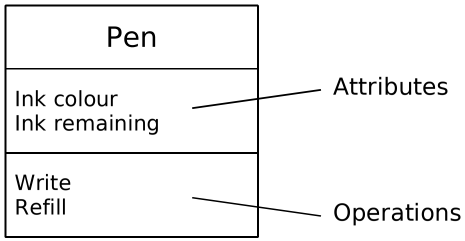
Objects that belong to a given class behave in the same way and are described by the same set of attributes

A class ‘specifies what an object is like’…

Another Example

Three objects, each in a different state

Only one class



Pen

Ink colour
Ink remaining

Write
Refill

C++ Classes

Objects and classes exist in any real-world problem that we might wish to solve through programming

Object-oriented languages like C++ allow us to work with these concepts directly

C++ classes are **abstractions** of real-world classes…

Class Definition Syntax

```
class  ClassName
{
  public:
    method prototypes

  private:
    instance variables
};
```

Functions to represent behaviour of the class, with implicit access to instance variables

Variables to represent attributes of the class, private so that users can’t access them directly

Don’t forget semicolon!

C++ Class Example: Header File

bank.hpp

```
namespace bank
{
  class Account
  {
    public:
      Account(const std::string&, int);
      string getIdentifier();
      int getBalance();
      bool deposit(int);
      bool withdraw(int);

    private:
      std::string identifier;
      int balance;
  };
}
```

constructor, responsible for creating instances of class

accessor method (queries object state)

mutator method (modifies object state)

C++ Class Example: Implementation

bank.cpp

```
#include "bank.hpp"

namespace bank
{
  Account::Account(const string& id, int bal)
  {
    identifier = id;
    balance = bal;
  }

  string Account::getIdentifier()
  {
    return identifier;
  }
  ...
}
```

`Account::` prefix tells compiler that these are methods of `Account`

Creating Instances & Calling Methods

```
Account myAccount("XYZ123", 250);

cout << myAccount.getBalance() << endl;
```

object name

method name

```
Account* accountPtr = new Account("ABC321", 0);

accountPtr->deposit(150);
```

‘arrow syntax’ when calling a method via a pointer

Exercise 5

Write a header file defining the class `Circle`.

Give your class definition

- An instance variable `radius` (type `double`)
- A constructor
- A `getRadius` method
- Methods `area` and `circumference`, returning the area and circumference, respectively, of the circle

Exercise 6

Implement the methods of the `Circle` class.

Note that the area and circumference of a circle of radius r are given by the formulae πr^2 and $2\pi r$.

You may assume that a constant called `M_PI` exists, representing π.

Improvements: Constructor Syntax & Method Inlining

```
namespace bank
{
  class Account
  {
    public:
      Account(const std::string& id, int b):
        identifier(id), balance(b) {}
      std::string getIdentifier() { return identifier; }
      int getBalance() { return balance; }
      bool deposit(int);
      bool withdraw(int);

    private:
      std::string identifier;
      int balance;
  };
}
```

Further Improvements: `const` Methods

What happens if we create an `Account` object as a `const` object and then try to call `getBalance` on it?

Tip: always make accessor methods `const`

Improvements: Constructor Syntax & Method Inlining

```
namespace bank
{
  class Account
  {
    public:
      Account(const std::string& id, int b):
        identifier(id), balance(b) {}
      std::string getIdentifier() const { return identifier; }
      int getBalance() const { return balance; }
      bool deposit(int);
      bool withdraw(int);

    private:
      std::string identifier;
      int balance;
  };
}
```

Operator Overloading

```
namespace bank
{
  std::ostream&
  operator << (std::ostream& out, const Account& account)
  {
    out << "Account " << account.getIdentifier()
        << ", GBP "   << account.getBalance();

    return out;
  }
}
```

```
Account myAccount("NDE1234", 1500);
cout << myAccount << endl;
```

Classes With Dynamically-Allocated Storage

```
class Array
{
  public:
    Array(): size(0), data(0) {}
    Array(int n): size(n) { data = new double[size]; }
    Array(const Array&);
    Array& operator=(const Array&);
    ~Array() { delete [] data; }

  private:
    int size;
    double* data;
};
```

We must provide a **copy constructor**, an **overloaded assignment operator** and a **destructor**...

Implementing a Copy Constructor

```
Array::Array(const Array& other): size(other.size)
{
  data = new double[size];

  for (int i = 0; i < size; ++i)
    data[i] = other.data[i];
}
```

Note use of a **const reference**...

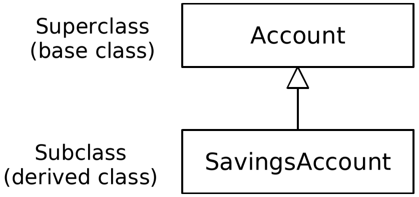
Overloading the Assignment Operator

```
Array& Array::operator=(const Array& other)
{
  if (this != &other) {
    delete [] data;

    size = other.size;
    data = new double[size];

    for (int i = 0; i < size; ++i)
      data[i] = other.data[i];
  }

  return *this;
}
```

Complicated! - better to use `vector`, `list`, etc…

Inheritance

One class can **specialise** another, inheriting its parent’s attributes and behaviour and introducing additional attributes / behaviour



Inheritance Example

```
class SavingsAccount: public Account
{
  public:
    SavingsAccount(const std::string& id, int bal):
      Account(id, bal), rate(0.05f) {}
    float getRate() const { return rate; }
    void setRate(float r) { rate = r; }
    void calculateInterest();

  private:
    float rate;
}
```

Summary

We have

- Looked at how C++ programs are compiled from multiple source files and libraries
- Introduced some of the fundamental concepts of object-oriented programming
- Considered how C++ classes are defined and used