Introduction to C++

David Marshall

School of Computer Science & Informatics
Cardiff University

CM2204

All Lecture notes, code listings on CM2204 Web page

C++

Bjarne Stroustrup, around 1986

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off"

- Designed (in 1979) by Bjarne Stroustrop to add object oriented features to C
  - C designed to be "close to the machine"
  - C++ designed to be "close to the problem to be solved"
- Recommended reading – *Thinking in C++, 2nd edition, Volume 1 (and partially Vol. 2)*, Bruce Eckel
  - http://www.mindviewinc.com/Books/downloads.html
  - Where possible, I'll use examples from the book

  Also see http://www.cs.cf.ac.uk/Dave/CM2204/:
  - Course Docs
  - Additional C/C++ Notes, Examples

From C to C++

Anything you can do in C++, you can do in C

- C gives you complete control
- C++ starts hiding things by providing higher level concepts
- Everything from C89 can be done in C++
  - But there is a C++ way and a C way

So What's different?

C++ adds new features

- Classes, inheritance, member functions
- References
- Templates
- Exceptions
- Overloading
- ...

Recap from Frank: C v C++ v Java

- Roughly Java: object-oriented with generics
  C++: object-oriented with templates
  C: procedural
- Object-oriented, procedural, functional, etc. is really a way of thinking, quite independent of programming language
  - Lisp can be OO, Java procedural, C functional...
- How best to think about a program?
  - Objects communicating with each other
  - Sequence of instructions
  - Transformations
- C++ supports object-orientation more than C
- Java has deliberate limitations to enforce cross-platform support and "cleaner" code

Java vs. C/C++ (Cont.)

C++

Full control, you decide what to do and how to do it

- C++ **trusts** that you know what you are doing
  - If you **do not**, you can **break everything**

Java

"Stick to my rules, and I do some of the hard work for you"

- Less understanding, less efficient, incomplete (machine details hidden, harder to adjust to specific problem, some things cannot be done, need for JNI)
- Java prevents you doing some things, hides and checks others
  - Maybe simpler, but always limited

Do you need / want the power/control of C/C++?

C++ Features

Procedural C
Global Functions
File-specific functions
Structs
Pointers (addresses)
Low-level memory access
C Preprocessor

Variables
Arrays
Loops
Conditionals

Classes
- Grouping of related data together
- With associated methods (functions)
'new' for object creation
'delete' for object destruction
Constructors, Destructors
Operator Overloading
Assignment operators
Conversion operators
Inheritance (sub-classing)
Virtual functions & polymorphism
Access control (private/public/protected)

Function Libraries
Standard functions
Custom libraries
O/S functions

Templates
(Generic classes)

Non-C features
e.g. References

Class Libraries
(+templated classes)
Standard library
Custom libraries
Platform specific libraries

Java Features

Procedural C
~~Global Functions~~
~~File-specific functions~~
~~Structs~~
~~Pointers (addresses)~~
~~Low-level memory access~~
~~C Preprocessor~~

Variables
Arrays
Loops
Conditionals

Classes
- Grouping of related data together
- With associated methods (functions)

'new' for object creation
~~'delete' for object destruction~~
Constructors, ~~Destructors~~
~~Operator Overloading~~
~~Assignment operators~~
~~Conversion operators~~ (toString()?)
Inheritance (sub-classing)
(ONLY) Virtual functions & polymorphism
Access control (private/public/protected)

~~**Function Libraries**~~
~~Standard functions~~
~~Custom libraries~~
~~O/S functions~~
Java Native Interface

~~**Templates**~~
'Generics' (weaker)

Non-C features
(ONLY) references

Class Libraries
(Standardised)
Collections
Networking
Graphics

C++ Programming: First steps and beyond

HelloWorld.cpp

```
#include <iostream>

int main(int argc, char** argv) {
    std::cout << "Hello World!" << std::endl;

    return 0;
}
```

Simple program but it lead to some issues and some new C++ concepts

Namespaces

- When using C, you need to be careful to avoid clashes between names of identifiers and functions
- C++ solves this problem by providing a mechanism to group related items into separate *namespaces*
- `iostream` library defines functions and objects in the `std` namespace, hence we need to prefix them by `std::`
- This can be cumbersome – we can instead expose all elements from the namespace:

time.cpp

```
#include <iostream>
using namespace std;

int main (int argc, char **argv) {
    cout << "Time is " << time(0) <<  endl;
    return 0;
}
```

Stream **output**

- The `iostream` library defines an object `cout` for output to the console/command line
- Can output different types (similarly to `toString` from Java)
- Can include various formatting modifiers

Stream2.cpp (From Thinking in C++):

```
#include <iostream>
using namespace std;

int main() { // Specifying formats with manipulators:
  cout << "a number in decimal: "
       << dec << 15 << endl;
  cout << "in octal: " << oct << 15 << endl;
  cout << "in hex: " << hex << 15 << endl;
  cout << "a floating-point number: "
       << 3.14159 << endl;
  cout << "non-printing char (escape): "
       << char(27) << endl;
}
```

Stream **input**

- The `iostream` class defines the `cin` object to get input from the **console/command line**

Numconv.cpp (From Thinking in C++):

```
#include <iostream>
using namespace std;

int main() {
  int number;
  cout << "Enter a decimal number: ";
  cin >> number;
  cout << "value in octal = 0"
       << oct << number << endl;
  cout << "value in hex = 0x"
       << hex << number << endl;
}
```

Standard C++ **string** class

- Character arrays in C are a little cumbersome:
  - Fixed size
  - Copying & concatenating
- C++ provides a standard `string` class (similar to Java)

HelloStrings.cpp (From Thinking in C++):

```
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1, s2; // Empty strings
  string s3 = "Hello, World."; // Initialized
  string s4("I am"); // Also initialized
  s2 = "Today"; // Assigning to a string
  s1 = s3 + " " + s4; // Combining strings
  s1 += " 8 "; // Appending to a string
  cout << s1 + s2 + "!" << endl;
}
```

The **vector** class

- C++ also includes a container that is more flexible than arrays — `vector`.
  - Similar to the `Vector` class in Java
- To access **element** `i` of a `vector a`:
  - in C++: `a[i]`
  - cf. in Java: `a.get(i)`
- Uses **templates** (similar to *generics* in Java) to allow elements of any type to be stored

vector example

IntVector.cpp (From Thinking in C++):

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Assignment
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
}
```

C++ vector Modifiers

Some modifiers:

- `assign` — Assign vector content
- `push_back` — Add element at the end. See IntVector.cpp above.
- `pop_back` — Delete last element
- `insert` — Insert elements
- `erase` — Erase elements
- `swap` — Swap content
- `clear` — Clear content

See text books and www.cplusplus.com/reference/vector/vector/ for full details.

Pointers & references

- Pointers work in C++ as they do in C
- C++ also adds *references*, which behave similarly, except:
  - References cannot be reassigned, pointers can;
  - Pointers can point to NULL, references can't;
  - You can perform arithmetic with pointers, but not references;
  - A few other more subtle differences.
- See following simple examples

Passing pointer by Address

PassAddress.cpp (From Thinking in C++):

```
#include <iostream>
using namespace std;

void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 56;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
}
```

Swapping Two Pointers

swap.cpp:

```
#include <iostream>
using namespace std;

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

int main() {
        int x = 1;
        int y = 5;
        cout << x << "\t" << y << endl;
        swap(x,y);
        cout << x << "\t" << y << endl;
}
```

Passing pointer by Reference

PassReference.cpp (From Thinking in C++):

```
#include <iostream>
using namespace std;

void f(int& r) { // Expects a reference
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x); // Looks like pass-by-value,
        // is actually pass by reference
  cout << "x = " << x << endl;
}
```

(Recap) C structs

- Structs in C group data together, e.g.

```
struct Time {
  int hour;
  int min;
  int sec;
}
```

- Use `.` to access members of a `struct` as a object
- Use`->` to access members of a `struct` via a pointer
- Common to define `typedef struct XX` to avoid tedious typing of `struct XX` each time you need the `struct`.

Simple struct Example

SimpleStruct.cpp

```
struct Structure1 {
  char c;
  int i;
  float f;
  double d;
};

int main() {
  struct Structure1 s1;
  s1.c = 'a'; // Select an element using a '.'
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
}
```

Simple typedef struct Example

SimpleStruct2.cpp

```
typedef struct {
  char c;
  int i;
  float f;
  double d;
} Structure2;

int main() {
  Structure2 s1;
  s1.c = 'a';
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
}
```

Simple typedef struct Pointer Example

SimpleStruct3.cpp

```
typedef struct Structure3 {
  char c;
  int i;
  float f;
  double d;
} Structure3;

int main() {
  Structure3 s1;
  Structure3* sp = &s1;
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;

}
```

On to C++ Structs: Bank Account Example

- BankAccountCStruct.h & BankAccountCStruct.cpp define a structure that represents a bank account using a C style struct
- Note:
  - We've defined functions to operate on a bank account
  - Syntax is a little awkward – every function needs the pointer to the bank account to be passed as an argument
  - Potential for name clashes
- First step to address these problems is to move functions within the struct – then they cannot clash

Bank Account Example: C Style Header

BankAccountCStruct.h

```
#include <string>

typedef struct CBankAccountTag {
        float balance;   // Account balance
        std::string name; // Account name
} BankAccount;

void initialise(BankAccount* b, std::string name);
void deposit(BankAccount* b, float amount);
void withdraw(BankAccount* b, float amount);
void transfer(BankAccount* from, BankAccount* to, float
    amount);
```

Bank Account Example: C Code

BankAccountCStruct.cpp

```
#include "BankAccountCStruct.h"

void initialise(BankAccount* b, std::string n) {
        b->name = n;
        b->balance = 0;
}

void deposit(BankAccount* b, float amount) {
        b->balance += amount;
}

void withdraw(BankAccount* b, float amount) {
        b->balance -= amount;
}

void transfer(BankAccount* from, BankAccount* to, float
    amount) {
        withdraw(from, amount);
        deposit(to, amount);
}
```

Bank Account Example: C++ style `struct` header

BankAccountCppStruct.h

```
#include <string>

struct BankAccount {
  float balance;  // Account balance
  std::string name; // Account name

  void initialise(std::string name);
  void deposit(float amount);
  void withdraw(float amount);
  void transfer(BankAccount& to, float amount);
};
```

Note:

- **No need** for a `typedef` of the structure.
- **No name clashes**: *e.g.* `BankAccount::deposit()`
- **No need** to pass **pointer** for `BankAccount`

Bank Account Example: C++ style struct: implementation

BankAccountCppStruct.cpp

```
#include "BankAccountCppStruct.h"

void BankAccount::initialise(std::string n) {
  this->name = n; // Can refer to members via this pointer
      ...
  balance = 0; // ... or implicitly
}

void BankAccount::deposit(float amount) {
  balance += amount;
}

void BankAccount::withdraw(float amount) {
  balance -= amount;
}

void BankAccount::transfer(BankAccount& to, float amount) {
  withdraw(amount);
  to.deposit(amount);
}
```

Notes on BankAccountCppStruct.cpp

- **Scope resolution operator** `::`
  (*e.g.* `BankAccount::initialise(std::string n)`

Scope resolution operator

Used to qualify hidden names so that you can still use them. You can use the unary scope operator if a namespace scope or global scope name is hidden by an explicit declaration of the same name in a block or class

- `this` keyword denoting the addres/pointer of the current object (instance of `struct BankAccount`)
  e.g. `this->name`

Bank Account Example: C++ style `struct` usage

BACStructTest.cpp

BACStructTest.cpp

```
#include "BankAccountCppStruct.h"
#include <iostream>
using namespace std;

int main() {
  BankAccount a, b;
  a.initialise("Stuart");
  b.initialise("Bob");
  a.deposit(5000);
  cout << a.name << " " << a.balance << endl;
  cout << b.name << " " << b.balance << endl;
  b.deposit(50000);
  cout << a.name << " " << a.balance << endl;
  cout << b.name << " " << b.balance << endl;
  b.transfer(a, 40000);
  cout << a.name << " " << a.balance << endl;
  cout << b.name << " " << b.balance << endl;
}
```

Implementation hiding

- Ideally we would like to control access to the members of the struct
- E.g. suppose our bank account has a member variable, `transactionCount`, to count the number of transactions:

```
void BankAccount::deposit(float amount) {
balance += amount;
transactionCount++;}
```

We want to prevent client code bypassing this:

```
a.balance += 5000;
```

- By default, everything in a struct is available to be accessed by anyone

C++ access control

- C++ defines three keywords to restrict access `public`, `private` and `protected`
  - `public` denotes that the member is available to all other code
  - `private` denotes that the member is only available within the struct
  - `protected` relates to inheritance – later in module
  - `friend` access is also possible – **not covered** in CM2204, but similar idea to package access in Java

Adding access control

- We easily could make the data members in our `BankAccount` `struct` `private` to prevent access.
- Instead, we should define it as a **class**, using the `class` keyword (which is only in C++) instead of `struct`
- Only one **difference** between `class` and `struct`:
  - **Default access** in `struct` is *`public`*
  - **Default access** in `class` is *`private`*
- **Best practice** to (generally) **only** use `struct` as used in **C**, use `class` for **anything** else in **C++**

Functions in C++: overloading and inline

- In C++ (**but not C**), functions are identified by their **name and** the **types** of their **parameters** (similar to Java):

```
int multiply(int a, int b) { return a * b; }
double multiply(double a, double b) { return a * b; }
```

  - Compiler finds matching function (converting types if necessary) – see overload.cpp
  - Be careful!

- **Inline** functions act as normal functions, but without the overhead of a function call:

```
inline int multiply(int a, int b) { return a * b; }
```

  - Useful for small, fast functions
  - Only advice – the compiler may ignore the instruction

Simple Function Overload Example: overload.cpp

overload.cpp

```
#include <iostream>
using namespace std;

int multiply(int a, int b) {
        cout << "In multiply(int a, int b)" << endl;
        return a * b;
}
double multiply(double a, double b) {
        cout << "In multiply(double a, double b)" << endl;
        return a * b;
}

int main() {
        cout << multiply(5, 4) << endl;
        double x = 0.5;
        double y = 2.0;
        cout << multiply(x, y) << endl;
        cout << multiply(0.5f, 2.0f) << endl;
}
```

On to the Lab Class:

After this handout & the following lab, you should:

- Be able to use input/output streams in C++;
- Understand the purpose of namespaces in C++;
- Be able to read and write text files in C++;
- Be familiar with the `string` and `vector` classes and their C equivalents;
- Understand the terms *operator overloading* and *references*.
- Understand the difference between `structs` in C and `structs` and `classes` C++;
- Be able to implement a simple class with member functions in C++;
- Use `public` and `private` to hide implementation;
- Understand the terms *function overloading* and *inline functions*.