UNIX and C Programming (COMP1000)

Lecture 10: C++

Updated: 29th July, 2015

Department of Computing
Curtin University

Textbook Reading (Hanly and Koffman)

For more information, see the weekly reading list on Blackboard.

- **Chapter 15: On to C++**
- C++ is introduced very briefly by Hanly and Koffman.
- For more information, look for a dedicated C++ book.
  - e.g. *Thinking in C++*, by Bruce Eckel (freely available).

Outline

C++ Overview

- This lecture will briefly introduce C++ from a Java(ish) perspective.
- However, C++ is much older than Java.
- C and C++ co-evolved.
- C++ is much more complex than C.

Commonalities with C

- C++ uses header files in the same way as C.
- C++ uses the C preprocessor (`#include`, etc.).
- C++ has the same primitive data types as C (`int`, `float`, etc., but also has a "`bool`" data type).
- C++ has the same basic control structures as C (`if`, `switch`, `for`, `while` and `do-while`).
- The `main()` function works the same way in C++ as in C.
- Like C, C++ lacks garbage collection.

Major Differences with C

- C++ has classes, with inheritance and polymorphism.
- C++ uses "`new`" and "`delete`" instead of "`malloc()`" and "`free()`"
- C++ has namespaces and a different set of libraries.
- C++ has exception handling (`try-catch`).
- C++ has operator overloading (which can radically alter the language!)

Filename Extensions

- The typical filename extensions are:
  - "`.cpp`" for a C++ source file.
  - "`.hpp`" for a C++ header file.
- The standard C++ header files *don't have extensions*:

```
#include <iostream>
```

Compiling

- Gcc can compile C++ as well as C:

```
[user@pc]$ gcc -c file1.cpp
```

```
[user@pc]$ gcc -c file2.cpp
```

```
[user@pc]$ gcc file1.o file2.o -o prog -lstdc++
```

- The ".cpp" extension tells gcc that it's dealing with C++ code.
- Other normal gcc options are supported as usual.
- "`-lstdc++`" tells gcc to link against the C++ library
- Alternatively, you can use "g++":

```
[user@pc]$ g++ file1.o file2.o -o prog
```

- Makefiles are used for C++, just as for C.

Namespaces

- A "namespace" is similar to a "package" in Java.
- Standard C++ functions and classes are generally in "`std`".
- Thus, you often have this right after your "`#include`"s:

```
using namespace std;
```

- Without this, you would have to prepend "`std::`" to a lot of things.

Scope Resolution Operator — "::"

- The "::" operator is used to refer to something inside a namespace or class (but not an object!)
- For instance, to call a **static** method:

```
c = ClassName::methodName(a, b);
```

- Also (if you don't have "`using namespace std;`"):

```
std::string str = "Hello world";
```

- However, to call a **non-static** method:

```
obj.methodName();
ptr->methodName();
```

  - "." and "->" mean the same thing as in C.
  - You may (or may not) have a pointer to the object.

Classes

- Classes in C++ are an extension to structs in C.
- They have methods (a.k.a. "member functions").
- They have public, private and protected fields and member functions.
- Conventionally:
  - The class and its members are *declared* in a header file.
  - Each member function is *defined* in the main source file.

Class Syntax

This would appear in "person.hpp":

```
class Person {
    public:
        Person();                    // Constructor
        Person(string inName, int inAge);
        ~Person();                   // Destructor

        int getAge() const;          // Accessor
        void setAge(int inAge); // Mutator
        ...  // Other methods

    private:
        string name;  // Private fields
        int age;
};
```

Member Function Definitions

This would appear in "person.cpp":

```
Person::Person(string inName, int inAge) {
    name = inName;
    age = inAge;
}

int Person::getAge() const {
    return age;
}

... // Other methods
```

- Note the use of "Person::" before each member.
- Note the "const" after getAge() — this means the method does not modify the object.

Creating and Destroying Objects

- In C++, the "`new`" keyword replaces `malloc()`:

```
Person* p;
p = new Person("Ralph", 7);
```

  - Allocates memory for a `Person` object.
  - Calls the constructor to initialise it.

- We can then manipulate the object as follows:

```
int a = p->getAge();
p->setAge(a + 1);
```

- Also, "`delete`" replaces `free()`:

```
delete p;
```

  - Calls the *destructor* to clean up.
  - De-allocates the memory.

Creating and Destroying Arrays

- The "`new`" keyword can also allocate arrays:

```
int* array;
array = new int[100];
```

- Such arrays must eventually be de-allocated with "`delete[]`" (*not* "`delete`"):

```
delete[] array;
```

Stack-Based Objects

We can also allocate objects on the stack:

```
void func() {
    Person p("Ralph", 7);    // Constructs p
    ...
    int age = p.getAge();
    p.setAge(age + 1);
    ...
}
```

- Declares a `Person` object on the stack (without "`new`").
- The constructor is called immediately.
- The destructor is called *automatically* at the end of the function.
- You use "`.`" to access fields and methods (no dereferencing).
- You can't do this in Java!

Destructors

- A destructor is the inverse of a constructor.
- With no garbage collection, C++ relies on destructors to clean up.
- There is only one destructor per class.
- It takes no parameters and returns nothing.
- Its role is to free resources (particularly memory) used by the object, before the object itself is removed from memory.

Static Methods

- C++ supports static fields and methods.
- A static field/method is shared by all instances of the class.
- This is same meaning as in Java (*not* C!)
- For example:

```
class Person {
    public:
        ...
        static void staticFunc(int param);
        ...
};
...
Person::staticFunc(10);
```

Member Access Operators

- C++ has 3 operators for accessing members of objects, classes and namespaces: "`.`", "`->`" and "`::`".
- You need each one in different circumstances:

Operator	Example	Used when...
`.`	`obj.method()`	`obj` is an object.
`->`	`ptr->method()`	`ptr` is a pointer to an object.
`::`	`cls::method()`	`cls` is a class or namespace.

A Note on Java

- Java has no equivalent to the "`.`" operator in C/C++ (because you can't have an object without a reference).
- Confusingly, Java uses the symbol "`.`" to replace "`->`" and "`::`".

The this Pointer

- As in Java, "this" is a pointer to the current object.
- Can be used to distinguish between fields and local variables or parameters.
- We could write our Person constructor as follows:

```
Person::Person(string name, int age) {
    this->name = name;
    this->age = age;
}
```

Inheritance

Classes can inherit/extend other classes, as follows:

```
class Employee : public Person {
    public:
        Employee();
        Employee(string name, int age, int salary);
        ~Employee();
    ...
};
```

However:

- Classes don't have to inherit from anything (unlike in Java).
- C++ supports multiple inheritance (use sparingly, if at all):

```
class SubClass : public SuperClassA,
                 public SuperClassB {
...
```

Virtual Methods

In C++, you can't override a method unless it's "`virtual`":

```
class Person {
    ...
    virtual bool equals(Person* p) const;
    ...
};

class Employee : public Person {
    ...
    virtual bool equals(Employee* e) const;
    ...
};
```

- `Employee::equals()` overrides `Person::equals()`.
- In Java, *all* methods are implicitly virtual.

Calling a Superclass Method

We need to call `Person::equals()` *inside* `Employee::equals()`

- In Java, we would do this:

```
if(super.equals(var) && ...) {        // Java
```

Java keyword

- However, C++ has no "`super`" keyword.
- Instead, you refer to the superclass's name:

```
if(Person::equals(var) && ...) {      // C++
```

superclass name

(This looks like a static method call, but C++ is smart enough to know otherwise.)

Calling a Superclass Constructor

- We also need to call `Person`'s constructor from within `Employee`'s constructor.
- C++ has a different syntax for this:

```
Employee::Employee(string inName, int inAge,
    int inSalary) : Person(inName, inAge)
{
    salary = inSalary;
}
```

- The super-constructor call is highlighted.
- `Person`'s constructor is called like a function, but *before* the first brace (and after a colon).

Pure Virtual (Abstract) Methods

- A virtual function is "pure" if it has no definition.
- A "pure" virtual function *must* be overridden.
- This is equivalent to an abstract method in Java (but there's no "abstract" keyword).
- An abstract class is any class containing a pure virtual function.

```
class Person {  // Abstract class
    ...
    virtual float calcTax() = 0;    // Pure virtual
    ...
};
```

Input and Output

C++ has a radically different syntax for I/O operations:

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

int main() {
    int num1, num2;

    cout << "Enter two integers: ";
    cin >> num1 >> num2;

    cout << "Sum: " << (num1 + num2) << endl;
    return 0;
}
```

- `cin` and `cout` are objects defined in the standard "`iostream`" library.

Reading and Writing Files

- The `fstream` library defines classes for reading and writing files.
  - `ifstream` — an input file stream.
  - `ofstream` — an output file stream.
- The constructor of ifstream/ofstream opens a given file.
- The destructor closes it.
- Read using the » operator (like `cin`).
- Write using the « operator (like `cout`).

Writing Files — Example

Write the integers 1–100 to file.txt:

```
#include <fstream>

int main() {
    ofstream* file = new ofstream("file.txt");

    for(int i = 0; i < 100; i++) {
        *file << i << endl;
    }

    delete file;
    return 0;
}
```

Writing Files — Alternative Example

Alternatively:

```
#include <fstream>

int main() {
    ofstream file("file.txt"); // Stack-based object

    for(int i = 0; i < 100; i++) {
        file << i << endl;
    }

    return 0;
}
```

- Now, `file` is an object on the stack — no pointers.
- The destructor is called automatically at the function's end.

References

- A C++ "reference" is a pointer that is *automatically dereferenced*.
  - When used, C++ automatically puts a * in front of it.
- References are set once (when declared). You can't change where they point.

```
int x = 10;
int& y = x; // y is a reference to integer x.

y = 20;     // Sets x to 20.
```

- Here, y is a reference to x.
- The & in the declaration means "reference-to", not "address-of".
- But what's the point?

Reference Parameters

- Functions (and methods) can take references as parameters:

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

...
int x = 10, y = 20;
swap(x, y);
```

- Be careful here — it's not obvious that `x` and `y` are being passed by reference!

Auto Pointers

- "Auto pointers" are another special type of pointer.
- *Only one* auto pointer can point to a given object!
  - If you try to copy the pointer, the original pointer will be wiped.
- Auto pointers are implemented by the class "`auto_ptr`".

```
#include <memory>
...
auto_ptr<Employee> ptr(new Employee());
```

This declares an auto pointer called "`ptr`", pointing to a new `Employee` object.

- When ptr disappears, the Employee object will be automatically deleted.

More Smart Pointers

- Auto pointers are one example of "smart" pointers.
- Smart pointers are special objects that represent pointers and make them safer or more convenient.
- The Boost library (not quite part of C++) defines several different types.
- Some are designed for arrays.
- Some are designed to allow multiple pointers to a single object.
- Auto pointers have now been deprecated in standard C++ by "unique pointers".

Templates

- Templates in C++ are similar to Java's generics (but the underlying implementation is very different).
- Templates let you write generic code that can handle any data type.
- This is especially useful for containers.
- Much of the C++ API uses templates (e.g. auto pointers).
- Both template functions and template classes are possible.

Template Declarations

- A template (function/class) begins with this line:

```
template <class A, class B, class C>
```

- This is followed by a normal(ish) function or class definition.
- The first line provides the "template parameters": `A`, `B` and `C`.
- These can be used within the function/class as ordinary data types.

Template Functions

A template function might look like this:

```
template<class T>
T square(T num) {
    return num * num;
}
```

Here, `square()` is a template function:

- When called with an `int`, it will return an `int`.
- When called with a `float`, it will return a `float`.
- For each data type you use, the compiler generates a separate copy of the function.
- When called with something that can't be multiplied, the compiler will output an error.

Template Classes (1)

A template class might look like this:

```
template<class T>
class Secret {
    public:
        Secret(T inObj, string inPassword);
        ~Secret();
        T getObj(string inPassword);

    private:
        T obj;
        string password;
};
```

- The "Secret" template class stores an arbitrary piece of data.
- The class itself doesn't care what the type is.

Template Classes (2)

We can instantiate the "Secret" template class as follows:

```
Secret<int> *s1 = new Secret<int>(5, "pass");
Secret<string> *s2 =
    new Secret<string>("spaceballs", "12345");

int a    = s1->getObj("pass");
string b = s2->getObj("12345");
```

- This creates two objects, pointed to by `s1` and `s2`.
- It also creates two versions of the class — one for `int`s and one for `string`s.
- The `getObj()` method returns the appropriate type.

Templates vs. Generics

- C++'s templates and Java's generics are not quite the same.
- Templates work by creating different copies for different types.
- Generics work by automatically inserting safe typecasts.
- They have similarities:
  - Both let you re-use code for different data types.
  - Both let you avoid gratuitous, unsafe typecasting.
- And differences:
  - Templates apply to *any* data type; generics apply only to objects.
  - Templates allow compile-time calculations ("metaprogramming").

The Standard Template Library (STL)

- C++ provides a number of standard template classes.
- These implement most commonly-needed abstract data types: lists, sets, maps, stacks, queues, iterators, etc.
- For example, `vector` is a resizeable list:

```
vector<int> vec;     // A vectors of ints

for(int i = 0; i < 10; i++) {
    vec.push_back(i);
}

for(int i = 0; i < 10; i++) {
    cout << vec[i] << endl;
}
```

Method/Function Overloading

- C++ allows for function and method overloading (like Java, but unlike C).
- Multiple functions/methods can have the same name, but different parameter lists.
- For example:

```
int readInt() {
    ...
}

int readInt(string prompt) {
    ...
}
```

The compiler will determine which method to call based on the parameters.

Default Arguments

- C++ lets you set "default" values for function parameters:

```
int readInt(string prompt = "Enter a value: ") {
    ...
}
```

You can call this function with or without a parameter:

```
int x = readInt();
int y = readInt("Enter y: ");
```

When no parameter is given, the default is used.

- You can do this for several parameters:

```
void func(int w, float x = 1.0, int y = 2) {...}
```

- Default arguments must come last in the parameter list.

Operator Overloading (1)

- In C++, it is possible to define different meanings for operators (+, *, ==, &&, ->, [], etc.).
- Instead of an `equals()` method for the Person class, we could define the == operator:

```
class Person {
    public:
        ...
        bool operator==(const Person& other) const;
        ...
};
```

- Here, we effectively have a method with the name "`operator==`".
- This will be called when you write "`person1 == person2`" (if `person1` and `person2` are both `Person` objects).

Operator Overloading (2)

- The applications of operator overloading are endless.
- It's how the « and » operators work with `cin` and `cout`.
- It's also heavily used by auto pointers and vectors.
- There are very few limits to what you can do.
- Use with care! (Gratuitous operator overloading makes your code **extremely unreadable**.)

Wrapping Up

- This has just been a taste of C++.
- There are still more features and many more nuances.
- There are also many other languages that follow in the footsteps of C.
- Some of these include: Objective C, Java, C#, D and Go.
- Sadly, this will not be in the exam!

That's all from UCP

- Hopefully you've understood most of it!
- Make sure you finish all the pracs.
- Good luck!