Preparing for the Programming with C++ exam

Christos Kloukinas

1 Major Differences between Java and C++

- C++ gives you access to the REAL OBJECTS, not just references (pointers) to them!!!!!!!
- call-by-reference (session 1 and since)

  Java supports only call-by-value, thus copying the arguments into new, local variables of the function.

  ```
  void foo( vector<double> v ) { ... }
  void foo( vector<double> &v ) { ... }
  void foo( const vector<double> &v ) { ... }
  ```
- function/operator overloading (session 3)

  Java doesn't allow **operator** overloading (it does allow method overloading though).

  ```
  void foo( int i ) { ... }
  void foo( double d ) { ... } // legal in both
  bool operator==( const Point &p ) { ... } // legal only in C++
  ```
- genericity or template classes (sessions 4–6)

  Java doesn't (well, didn't…) support genericity.

  ```
  template <typename T>
  class vector<T> {
    public:
      vector();
      typedef T *iterator;
      ostream &printall(ostream &o);
  }
  template <typename T>
  vector<T>::vector() {...}
  template <typename T>
  ostream &vector<T>::printall(ostream &o) {
    typename vector<T>::iterator it;
    for (it = begin(); it != end(); ++it)
      o << *it;
    return o;
  }
  ```
- memory management

  In Java objects are always allocated on the heap and reclaimed automatically by the garbage collector.

  - local allocation of objects (sessions 1–2 and since, sessions 9–11)

    ```
    void foo () {
      Point p; // local allocation on the stack - deleted automatically
               // when foo exits
    }
    ```
  - pointers (sessions 5 and 6)

    ```
    Point p;
    Point *pp = &p;
    Point *pp2 = pp + 3; // POINTER ARITHMETIC!!! - NOT IN JAVA!
    ```
  - dynamic allocation (sessions 8–9)

    Done with `new` as in Java, allocating objects on the heap but now it's the programmer's responsibility to reclaim the memory (with `delete`) when the objects are no longer needed.
- multiple inheritance (session 7)

  In Java one can inherit only from one class but can implement many interfaces (abstract classes).

  Why? To avoid the ambiguities arising in C++ with multiple inheritance, where two super-classes may have the same member variables/functions.

2 Things you should be able to do

- Write simple classes and functions in C++.
- Use the containers and iterators of the Standard Template Library to write more compact programs.
- Understand the difference between call-by-value and call-by-reference.
  EXTREMELY IMPORTANT!!!

  Call-by-reference essentially passes a reference (alias) for the original parameter, while call-by-value creates a new, local variable and copies the original parameter inside that one.

  If one changes the argument inside the function, then the original parameter will change as well if we use call-by-reference but it will not change if we use call-by-value.
- Appreciate the various meanings of `const` in C++, and know when to use them.

  `const double pi = 3.14156;` A constant object

  `void foo (const Point &p) {}` Used in call-by-reference to state that the original parameter will not be changed.

  `void member_foo () const {}` Used at the end of *member* functions to state that they don't change the state of the object.

  `const Point & foo () {}` States that you get a reference to some object but you cannot change it - similar to the call-by-reference example.
- Read programs using overloaded operators, by identifying which methods or independent functions are called.

  To identify which overloaded method/function is called we need to look at the types of the arguments - are they `int`, `double`, `Point`, ...?

  Sometimes, an argument of type `T1` can be transformed automatically by the compiler to another type `T2`. For example, an `int` can be transformed to a `double`. We need to consider these transformations as well to understand which overloaded function is called.
- Define overloaded operators for new types.

  Two ways to do it - either inside or outside the class (NOT BOTH!)

  ```
  class Point {
    public:
      bool operator==(const Point &o) {} // Inside the class
  };

  bool operator==(const Point &p1, const Point &p2) {} // Outside the class
  ```

  Inside the class the overloaded operator is a member function, so its *implicit* first argument is `*this`, the object itself. Thus, the statement `p1 == p2` is equivalent to `p1.operator==(p2)`.

  Outside the class, the overloaded operator is a normal function so we must explicitly declare both parameters. Thus, the statement `p1 == p2` is now equivalent to `operator==(p1, p2)`.

  When overloading `operator<<` or `operator>>` then we can only use the external overloading because the first parameter must be a stream, not the object we want to print/read!!!

3 More things you should be able to do

- Distinguish between objects and pointers, and what can be done with them.

  A usual problem for Java programmers, since in Java one can only manipulate references (i.e., pointers) not the objects themselves.

  ```
  derived_point  d;
  point          p = d; // p gets a SLICE of d
  derived_point *dp = &d;
  point         *pp = dp; // NO SLICING HERE
  ```

  See session 6, slides (6-17)–(6-21) to see how pointers and containers are used together.

- Know how to use static, local, dynamic and temporary allocation, appreciating their properties and distinctive features.

  Session 8:

	Storage allocated, constructor called	**Destructor is called, storage is reclaimed**
static object	when the program starts	when the program terminates
local object	when the declaration is executed	on exit from the function or block IN THE OPPOSITE ORDER OF DECLARATION!
free object	when `new` is called	when `delete` is called
subobject	**BEFORE** the containing object is created	**AFTER** the containing object is destroyed

- Understand the properties of subobjects (objects that are fields of other objects).

  ORDER OF INITIALISATION:

  Initialization is done in the following order:

  1. Constructors for base classes
  2. Members (in order of declaration)
  3. Body of constructor

  WHY? Because super-classes & member objects must exist when the body of the constructor is called, since the latter refers to the former!

  NOTE: The order in which the initialisation list is given has no effect!

  Destruction is done in the OPPOSITE order:

  1. Body of destructor
  2. Members (in OPPOSITE order of declaration)
  3. Destructors for base classes

  WHY? Because super-classes & member objects must still exist when the body of the destructor is called, since the latter refers to the former!

- Use inheritance, method redefinition and abstract classes in C++.

  `class B : public class A {...};` `B` inherits from `A`

  To redefine a method, that method MUST be declared as `virtual` in the super-class!

  To declare a method as abstract (and thus the class which contains it) you must:

  1. Declare it as `virtual`
  2. Define it to be equal to ZERO: `virtual void foo() = 0;`

- Write generic classes and functions in C++.

4 Even more things you should be able to do

- Use multiple inheritance in C++, knowing how to specify replicated versus repeated/shared inheritance (`virtual`).

  See session 7 - `Radio` (replicated multiple inheritance of `Storage`) versus `Painter` (repeated/shared multiple inheritance of `Window`).

  Slides (7-18)–(7-22) for calling a super-class' method only once are VERY IMPORTANT!

- Explain what the automatically generated constructors, destructors and assignment operators do, when they are inadequate, and if so how they should be replaced.

  – default constructor `A()`:
    Created if you don't define any constructor.
    Does NOTHING (its body is `{}`).
  – default copy constructor `A(const A &a)`:
    Created if you don't define it yourself.
    Initialises member variables using the values (copying them) of the other object.
  – default assignment operator `A & operator=(const A &a)`:
    Created if you don't define it yourself.
    Assigns to the member variables the values of the other object.

  - default destructor `~A()`:
    Created if you don't define it yourself.
    Does NOTHING.
    It's NOT a virtual destructor, see slides (8-10)–(8-11) for why we may need a virtual destructor.

  The destructor should be redefined if you have pointers as member variables or you need to do something fancy when the object is deleted.

  GANG OF THREE rule: If you need to define a destructor then you most probably need to define a copy constructor and an assignment operator as well, e.g., to perform *deep* copying, see slides (8-16)–(8-26).

- Use the exception syntax of C++.

  See session 10 as well as the lab & lab solutions for session 10.

- Use the "resource acquisition is initialization" technique to safely release resources, even in the presence of exceptions.

  MUST know how to use `auto_ptr<T>` AS WELL AS how to define it yourself!!!!!

5 HOW????

Study the handouts and labs, TRY TO TEST THE CODE YOURSELF so as to really understand what it does. Look at the coursework solutions - again, try the code out, possibly changing it to see what it does. Have a look at past exams as well! Try to answer the questions. Experiment with things you have a problem understanding. For example, to verify the order of constructors/destructors write something like this:

```
$ g++ -Wall -pedantic -ansi -g constructor-example.cc -o constructor-example
$ ./constructor-example
I'm the constructor of B - a super-class
I'm the constructor of A - a member object
I'm the constructor of C - a sub-class
I'm the destructor of C - a sub-class
I'm the destructor of A - a member object
I'm the destructor of B - a super-class
$
```

Oh, yes, you can always post [1] questions on Moodle.

[1] And *answer* too! ;-)