Final Exam Answers – CS 246 Fall 2009

Instructor: Peter Buhr

December 17, 2009

These are not the only answers that are acceptable, but these answers come from the notes or class discussion.

1. (a) **3 marks** A constructor is a special member used to **perform initialization** to ensure an object is valid before use. It is **called by the compiler** immediately **after object allocation**.

   (b) **2 marks** Routine operators allow **implicit conversions** on **both operands** of an infix call.

   (c) **4 marks** Initialization involves a **newly allocated object with undefined values**. Assignment involves an **existing object that may contain previously computed values**. The mechanism used to implement each in C++ is the **copy constructor** and the **assignment operator**.

   (d) **2 marks** A *shallow*-copy **copies pointer values**. A *deep*-copy **copies the values referenced by pointers** (possibly recursively).

   (e) **4 marks**

```
1   struct T2;    // forward
    struct T1 {
1       T2 *t2;   // pointer, break cycle
1       T1();     // forward declaration
    };
    struct T2 {
        T1 t1;
    };
1   T1::T1() { t2 = new T2; } // can now see T2
    T1 t1;
```

```
1   struct T2;    // forward
    struct T1 {
1       T2 *t2;   // pointer, break cycle
1       T1( T2 *t2 ) : t2( t2 ) {};
    };
    struct T2 {
        T1 t1;
    };
.5  T2 t2;
.5  T1 t1( t2 );
```

2. (a) **2 marks** Type inheritance relaxes name equivalence by **aliasing the derived name** with **its base-type names**.

   (b) **1 mark** routine pointers

   (c) **5 marks**

   i. bp.f(); *// Base::f*

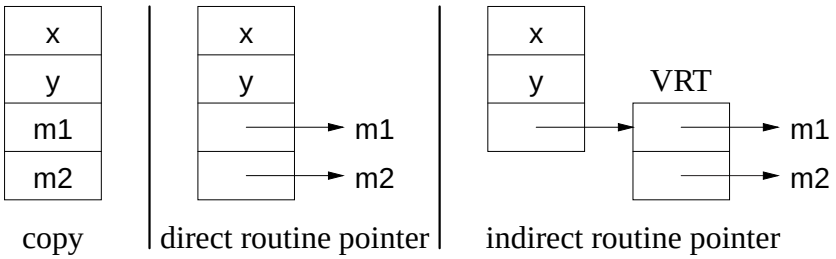
   ii. bp.g(); *// Base::g*

   iii. ((Derived &)bp).g(); *// Derived::g*

   iv. bp.Base::h(); *// Base::h*

   v. bp.h(); *// Derived::h*

   (d) **3 marks** Right picture



   (e) **3 marks** A *down cast* is a **dynamic check** to determine the **actual type an object** pointed to by a **polymorphic pointer/reference**.

3. (a) **2 marks** Routine template generalizes code across multiple types. Type template generalizes data structures across multiple types.

   (b) **2 marks** Nodes are either **copied into the container** or **pointed to from the container**.

   (c) **2 marks** An iterator **traverses a container** so knowledge about the container **implementation is hidden**.

   (d) **2 marks** `begin()` points at the first node/element of the container; `end()` points *after* the last node/element of the container.

   (e) **3 marks** qualification (`std::cout`), individual import (**`using`** `std::cout`), importing all (**`using namespace`** `std`)

4. (a) **2 marks** The compiler flag `-O2` controls the amount of optimization performed during compilation. This flag is not on all the time because it increases the cost of compilation.

   (b) **2 marks** Debugging is the process of determining why a program does not have an intended behaviour.

   (c) **2 marks** Control-flow error is incorrect transfer of control during execution. Data-flow error is incorrect computation of values during execution.

   (d) **1 mark** false

   (e) **2 marks** The `g++` compiler provides the `-MMD` flag to generate a dependency graph from the include files in a source file.

5. (a) **1 mark** *truth* is in the code

   (b) **1 mark** false

   (c) **2 marks** System modelling involves modelling a complex system in an abstract way to provide a specific description of how the system works.

   (d) **3 marks**

   - **sketch** out high-level design or complex parts of a system,
   - **blueprint** the entire system abstractly with high accuracy,
   - **generate** interfaces directly.

   (e) **2 marks** Association : a named conceptual/physical connection among objects.

   (f) **3 marks** Managed language hides aspects of the implementation, e.g., like memory management. An advantage is the reduction in low-level program tasks. A disadvantage is the inability to perform low-level operations for efficiency purposes.

6. (a) **1 mark** In *agile* development process, programmers often work in pairs.

   (b) **2 marks** A *design pattern* is a **common/repeated issue**; it can be a **problem or a solution**.

   (c) **2 marks** equivalence partitioning : partition all possible input cases into equivalence classes and select only one representative from each class for testing

   boundary value : test cases which are below, on, and above boundary cases

   (d) **2 marks** regression testing : test if new changes produce different effects from previous version of the system (diff results of old / new versions).

   performance testing : test if program achieves speed and throughput requirements.

   (e) **1 mark** false

7. **45 marks**

```
      class UTSImpl {
        protected:
          Printer &prt;
          NameServer &nameServer;
          BottlingPlant &plant;
          const unsigned int NumVendingMachines;
          const unsigned int MaxStockPerFlavour;
1         unsigned int shipment[ NUM_OF_FLAVOURS ]; // shipment received from bottling plant
1         unsigned int tracking, current;
1         bool hired;
1         vector<bool> history;                     // tracking history
1         unsigned int *vending;                    // used to hold a vending machine's inventory
1         VendingMachine **masterList;              // list of vending machines from name server
        public:
          // constructor, pickup, status, action as given in UTS
      }; // UTSImpl

1     class UTSeast : public UTS, private UTSImpl {    // as given

1     class UTSwest : public UTS, private UTSImpl {    // as given

      UTSImpl::UTSImpl( Printer &prt, NameServer &nameServer, BottlingPlant &plant,
          unsigned int numVendingMachines, unsigned int maxStockPerFlavour ) :
              prt( prt), nameServer( nameServer ), plant( plant ), tracking( 0 ), current( 0 ), hired( false ),
              NumVendingMachines( numVendingMachines ), MaxStockPerFlavour( maxStockPerFlavour ) {
1         prt.change( Printer::UTS, 'S' );
1         masterList = nameServer.getMachineList();
      } // UTSImpl::UTSImpl

      UTSImpl::~UTSImpl() {
1         prt.change( Printer::UTS, 'F' );
      } // UTSImpl::~UTSImpl

      unsigned int UTSImpl::pickup() {
1         hired = true;
1         history.push_back( false );
1         return tracking ++;
      } // UTSImpl::pickup

      bool UTSImpl::status( unsigned int tracking ) {
1         prt.change( Printer::UTS, 's', history[tracking] );
1         return history[tracking];
      } // UTSImpl::withdraw
```

```
1   void UTSImpl::action( int direction ) {
1     if ( ! hired ) return;
1       hired = false;
1       history[current] = true;                    // change tracking status
1       current += 1;

1       plant.getShipment( shipment );              // pick up shipment from bottling plant
1       unsigned int numLeft = 0;                   // calculate amount in shipment
1       for ( unsigned int i = 0; i < NUM_OF_FLAVOURS; i += 1 ) numLeft += shipment[i];
1       prt.change( Printer::UTS, 'P', numLeft );

        // Make a delivery to each vending machine in turn, so long as still have stock remaining
1       for ( unsigned int i = 0; numLeft > 0 && i < NumVendingMachines; i += 1 ) {
1           unsigned int index = direction != 0 ? direction - i : i;
1           vending = masterList[index]->inventory();   // obtain stock left from vending machine
1           prt.change( Printer::UTS, 'd', masterList[i]->getId(), numLeft );

            // Calculate how much of the shipment needs to be sent to the vending machine
1           for ( unsigned int j = 0; j < NUM_OF_FLAVOURS; j += 1 ) {
1               unsigned int difference = MaxStockPerFlavour - vending[j];
1               if ( difference > shipment[j] ) difference = shipment[j];
1               shipment[j] -= difference;
1               numLeft -= difference;
1               vending[j] += difference;
            } // for

1           prt.change( Printer::UTS, 'D', masterList[i]->getId(), numLeft );
1           masterList[i]->restocked();             // refilling complete
        } // for
    } // UTSImpl::balance

    UTSeast::UTSeast( Printer &prt, NameServer &nameServer, BottlingPlant &plant,
                      unsigned int numVendingMachines, unsigned int maxStockPerFlavour ) :
1       UTSImpl( prt, nameServer, plant, numVendingMachines, maxStockPerFlavour ) {
    } // UTSeast::UTSeast

1   unsigned int UTSeast::pickup() { return UTSImpl::pickup(); }
1   bool UTSeast::status( unsigned int tracking ) { return UTSImpl::status( tracking ); }
1   void UTSeast::action() { UTSImpl::action( 0 ); }

    UTSwest::UTSwest( Printer &prt, NameServer &nameServer, BottlingPlant &plant,
                      unsigned int numVendingMachines, unsigned int maxStockPerFlavour ) :
1       UTSImpl( prt, nameServer, plant, numVendingMachines, maxStockPerFlavour ) {
    } // UTSwest::UTSwest

1   unsigned int UTSwest::pickup() { return UTSImpl::pickup(); }
1   bool UTSwest::status( unsigned int tracking ) { return UTSImpl::status( tracking ); }
1   void UTSwest::action() { UTSImpl::action( NumVendingMachines - 1 ); }
```