CSE 333 Final Exam (2nd Midterm) Sample Solution 8/22/13

Question 1. (20 points) A little C++ hacking. On the following page, implement function `undup`. The input to this function is a list of strings that are not sorted in any particular order. The function should return a pointer to a newly allocated list of strings on the heap that is a copy of the original list, except that if the original list contains two or more adjacent copies of some string, those adjacent copies should be replaced by a single copy of that string. For example, if the input list contains

```
{"apple", "donut", "donut", "banana", "banana", "banana",
 "cherry", "cherry", "banana", "donut"}
```

then `undup` should return a pointer to a new `list<string>` containing the following:

```
{"apple", "donut", "banana", "cherry", "banana", "donut"}
```

Notice that a string might appear more than once in the result if it there are multiple copies of it in the original string separated by other strings.

For full credit, your function must scan the input list only once and may not use any other containers like lists, maps, or arrays. It may, of course, have whatever simple variables are needed, including strings.

Hints and reference information:

- Remember that a STL `list` is a linked list underneath, so you must use iterators to scan it; you can't use `[]` subscripts to access individual list elements.
- If `lst` is a STL `list`, then `lst.begin()` and `lst.end()` return iterator values of type `list<...>::iterator` that might be useful.
- If `it` is an iterator, then `*it` can be used to reference the item it currently points to, and `++it` will advance `it` to the next item, if any.
- Some useful operations on sequential STL containers, including `list`:
  - `c.clear()` – remove all elements from c
  - `c.size()` – return number of elements in c
  - `c.empty()` – true if number of elements in c is 0, otherwise false
  - `c.push_back(x)` – copy x to end of c
  - `c.push_front(x)` – copy x to front of c
- You are free to use the C++11 `auto` keyword, C++11-style `for`-loops for iterating through containers, and any other features of standard C++11, but you are not required to use these.

(Write your answer on the next page. You may remove this page from the exam if you wish.)

Question 1. (cont.) Complete the definition of function `undup` below. Some useful `#include` directives as well as a `using` directive have been provided for convenience. You should add any additional libraries or code that you need (although the sample solution only needed the ones given here).

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

// return a pointer to a new heap-allocated copy of lst
// where adjacent duplicate strings in lst are replaced
// by a single copy of that string.
list<string> * undup(const list<string> &lst)  {
  // Version 1: C++98-style iterators
  list<string> * ans = new list<string>();  // result
  string prev = "";     // most recent string added to ans
  for (auto it = lst.begin(); it != lst.end(); ++it) {
    if (*it != prev) {
      prev = *it;
      ans->push_back(*it);
    }
  }
  return ans;
}

// Version 2 with C++11 for-loop
list<string> * undup(const list<string> &lst)  {
  list<string> * ans = new list<string>();  // result
  string prev = "";      // most recent string added to ans
  for (auto item: lst) {
    if (item != prev) {
      prev = item;
      ans->push_back(item);
    }
  }
  return ans;
}
```

Question 2. (18 points) A little bit of class. Here is a tiny C++ class that holds a single integer as an instance variable.

```
class Int {
public:
  // constructors
  Int(): val_(0)      { cout << "cons "; }
  Int(int n)          { val_ = n; cout << "intcons "; }
  Int(const Int &v)
     : val_(v.val_) { cout << "copycons "; }

  // operations
  int get_val() {
    cout << "get_val ";
    return val_;
  }
  Int operator+(const Int &v) {
    cout << "op+ ";
    return Int(val_ + v.val_);
  }
  Int & operator=(const Int &v) {
    cout << "op= ";
    if (this != &v) { val_ = v.val_; }
    return *this;
  }
  Int & operator+=(const Int &v) {
    cout << "op+= ";
    val_ += v.val_;
    return *this;
  }

private:
  int val_;     // value stored in this Int
};
```

Answer the question about this class on the next page. You can remove this page for reference if you wish.

Question 2. (cont.) What output is produced when we execute the following program that uses the `Int` class defined on the previous page?

```
int main() {
  Int zero;
  Int one = zero + 1;
  cout << ";\n";
  Int two = zero;
  two += one + one;
  cout << ";\n";
  cout << two.get_val() + 1;
  cout << endl;
  return 0;
}
```

Output:

```
cons intcons op+ intcons ;  *

copycons op+ intcons op+= ;

get_val 3
```

***The `intcons` calls towards the end of the first and second lines are because an `Int` object is constructed from an `int` value in the `return` statement for `operator+`.**

In the first line, we allowed credit for an extra "`copycons`" output at the end, since the `zero+1` value is an `Int` that is used to initialize variable `one`, which would be done by a copy constructor However, as an optimization, the compiler removed the additional copy constructor call, and the output given above is what we observed when we ran the code on the CSE linux system.

Question 3. (18 points) Virtual madness. What output is produced when we run the following program? It does compile and run without errors.

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

class Base {
public:
  virtual void w() {      cout << "Base::w" << endl;   }
  virtual void x() { y(); cout << "Base::x" << endl;   }
          void y() {      cout << "Base::y" << endl;   }
  virtual ~Base()  {      cout << "Base::dtr" << endl; }
};

class A: public Base {
public:
  virtual void w() { x(); cout << "A::w" << endl;   }
          void y() {      cout << "A::y" << endl;   }
  virtual ~A()     {      cout << "A::dtr" << endl; }
};

class C: public A {
public:
  virtual void y() { cout << "C::y" << endl;   }
  virtual ~C()     { cout << "C::dtr" << endl; }
};

int main (int argc, char **argv) {
  Base *ptr = new C();
  ptr->y();
  cout << "-----" << endl;
  ptr->w();
  cout << "-----" << endl;
  ptr->x();
  cout << "-----" << endl;
  delete ptr;
  return 0;
}
```

Output:

```
Base::y
-----
Base::y
Base::x
A::w
-----
Base::y
Base::x
-----
C::dtr
A::dtr
Base::dtr
```

Question 4. (18 points) Below is the pseudo-code for a very simple TCP server that accepts connections from clients and exchanges data with them. However, this code doesn't work because it has structural errors. In particular, some functions are called at the wrong time or in the wrong place, but there may be other problems. Write in corrections below to show how the pseudo-code should be rearranged, changed, or fixed to have the proper structure for a simple server. Feel free to draw arrows showing how to move code around, but be sure it is clear to the grader what you mean.

You should assume that all functions always succeed – ignore error handling for this question. Further, assume that the first address returned by `getaddrinfo` works and we don't need to search that linked list to find one that does work. Also, ignore the details of parameter lists – assume that all the "…" parameters are valid and appropriate.

```
int main(int argc, char **argv) {
  struct addrinfo hints, *rp;
  memset(&hints, 0, sizeof(hints));

  hints.ai_... = ...;    // specify values for options

  getaddrinfo(NULL, argv[1] &hints, &rp);

  //  ok to assume *rp is a valid address and will work here
  int fd = socket(rp->ai_family,
                  rp->ai_socktype, rp->ai_protocol);

  setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, ...);

  freeaddrinfo(rp);

  while (1) {

    bind(fd, rp->ai_addr, rp->ai_addrlen);

    fd = accept(fd, ...);

    listen(fd, SOMAXCONN);

    // talk to client as needed
      read(fd, ...);
      write(fd, ...);

    close(fd);

  }
  return EXIT_SUCCESS;
}
```

Question 4. Here is the corrected version of the pseudo-code. Major changes:

- **Use separate `int` file descriptors for the listening and client sockets.**
- **Move the code to bind and open the listening socket outside the loop so it is done once as part of the server initialization.**
- **Only close the client file descriptor inside the loop; leave the listening fd open so it can be used to accept the next client.**

The listen file descriptor should not be closed until the server exits, so that must be done outside the loop. However we didn't show any code to shut down the server, so if the `close(listen_fd)` operation was omitted, no points were deducted.

```
// pseudo-code for a network server
// assume miraculous behavior - all attempts to do something work
int main(int argc, char **argv) {
  struct addrinfo hints, *rp;
  memset(&hints, 0, sizeof(hints));
  hints.ai_... = ...;   // specify values for desired options
  getaddrinfo(NULL, argv[1] &hints, &rp); // assume *rp will work
  int listen_fd = socket(rp->ai_family,
                         rp->ai_socktype, rp->ai_protocol);
  setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, ...);
  bind(listen_fd, rp->ai_addr, rp->ai_addrlen);
  freeaddrinfo(rp);
  listen(listen_fd, SOMAXCONN);
  while (1) {
    int client_fd = accept(listen_fd, ...);
    // talk to client as needed
      read(client_fd, ...);
      write(client_fd, ...);
    close(client_fd);
  }
  close(listen_fd);
  return EXIT_SUCCESS;
}
```

Question 5. (12 points) Concurrency. Suppose we are executing a workload where each transaction requires 3 disk I/O operations with short bursts of CPU time at the beginning, at the end, and in between each pair of disk operations. A a single transaction (not drawn to scale) looks something like this:

CPU	I/O	CPU	I/O	CPU	I/O	CPU

Assume that each I/O operation takes 10 msec. (i.e., 0.010 sec.) and each burst of CPU time takes 10 microseconds.

(a) How many transactions per second can we perform if we execute transactions sequentially, where a new transaction can't start until the previous one is completed? You should give a "back-of-the-envelope" estimate – i.e., don't use a calculator (which is not allowed on the exam anyway). Just give a good answer accurate to a couple of significant digits. To help the grader, show enough of your work or give a brief justification so we can follow it.

The time for each transaction is 30.040 msec (3 I/O operations @ 10 msec each plus 4 CPU slices @ 0.01 msec each). The CPU time is overwhelmed by the I/O time, so, to a couple of significant figures, each transactions takes 30 msec. In one second we can do a bit more than 33 of these (33 * .030 sec = .99 sec).

(b) Now, suppose we use a multi-threaded implementation to execute as many transactions in parallel as possible. How many transactions per second can we execute now? You should assume that we have unlimited I/O resources so we can process as many I/O requests in parallel as desired, and you can assume that the concurrency does not add any measurable overhead or time needed to process the transactions.

Under these assumptions, the I/O operations do not delay any transactions, so the cost per transaction is 40 microseconds, or 0.00004 seconds. We can do 25000 of these in one second.

(These assumptions aren't totally realistic for actual systems since I/O bandwidth is not unlimited and there is some overhead for thread dispatch and switching plus handling I/O operations. But even with more realistic assumptions the speedup from concurrent execution can be dramatic.)

Question 6. (6 points) One of the summer interns has been told to implement automatic memory management for our new implementation of Java. The idea was to implement a garbage collector to automatically reclaimed unused, dynamically-allocated data. But the intern thinks it would be simpler to use reference counting, where the implementation keeps track of how many pointers refer to each piece of dynamically allocated data, and frees (deletes) any data whose reference count becomes 0.

The question is, will this work? Is reference counting a suitable substitute for automatic garbage collection? Give a brief technical justification for your answer.

No, not completely. If a dynamically allocated data structure contains cycles then all of the items in the cycle will have non-zero reference counts even if there are no other references to the data. A garbage collector would reclaim such data; a reference counting implementation won't (unless we use additional strategies like weak pointers).

Question 7. (6 points) Almost all of the time when we are programming with classes and subclasses, we use virtual functions and dynamic dispatching so that the actual types of the data objects determine which functions are executed. But there are occasional times where it makes sense to omit the `virtual` keyword and determine the actual function at compile-time, regardless of the run-time types of the data. Give two (brief) reasons why we might want to do this.

The two most common reasons:

(i) A non-virtual function call is slightly faster since there is no need to follow the pointers from the object to the vtable to the function code.

(ii) The use of non-virtual functions can guarantee that a call to a function `g()` will call a function that belongs to the class and not one that is later added in some subclass.

An even more obscure reason is that if a class contains no virtual functions at all then instances of that class will not contain a virtual function table pointer. That is sometimes needed to create a C++ data structure that matches the layout of an existing simple `struct` or other data structure.

Question 8. (2 free points!)

What is the answer to this question? (Write an interesting answer below.)

42

(Reference: *The Hitchhikers Guide to the Galaxy*, Douglas Adams, 1979.)

[Of course, all answers received full credit on this question.]