# **Exam 1 Solution**

*This is a closed book and closed notes test.* You are not allowed to have anything on your desk other than pencil and this exam paper during the test; this includes *calculators* or *electronic assistance* of any kind – ***especially smartphones***.

*You may not leave to go to the restroom.* Please go before the exam starts.

*You may not ask questions.* If something is confusing, write a note beside the question and explain your assumptions.

*You must show all of your work on this exam.* You will not be allowed to turn in additional sheets of paper.

*Read and sign the following statement.*  Failure to sign the statement will result in a **zero** on the exam.

*I have neither given nor received unauthorized assistance on this test. I have notified the proctor of any violations of the above policies.*

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **Problem** | **Score** |
| 1 | / 30 |
| 2 | / 30 |
| 3 | / 20 |
| 4 | / 20 |
| **Total** | **/ 100** |

*Points divided evenly among parts of a problem unless otherwise specified.*

1. [30 points] Answer the following questions about queues and stacks. In all cases, you should assume they are implemented using a singly-linked list. Class declarations shown below. You may assume that T is an object that’s fully implemented, with appropriate constructors, destructors, copy constructors, and no pure virtual functions.

|  |  |
| --- | --- |
| #include “T.h” // T definition  class Queue {  //omitted implmentation  public:  void push(T &t);  T& peek();  void pop();  bool empty();  }; | #include “T.h” // T definition  class Stack {  //omitted implementation  public:  void push(T &t);  T& peek();  void pop();  bool empty();  }; |

1. [15 points] Fill in the table below with a **brief** description in words of what each function in the Queue and Stack interface mean according to the discussion from lecture and the zyBook. Describe them in terms of an underlying single-linked list implementation. Do not show code!

|  |  |  |
| --- | --- | --- |
| Function | Stack | Queue |
| void push(T &t); | **Add t to front of list.** | **Add t to tail of list.** |
| T& peek(); | **Return reference to front of list.** | **Return reference to front of list.** |
| void pop(); | **Remove item from head of list.** | **Remove item from head of list.** |
| bool empty(); | **Return if list is empty (head==nullptr)** | **Return if list is empty (head==nullptr)** |

1. [15 points] Assuming that the Queue and Stack classes are fully implemented (your answer should not implement them) with appropriate constructors, destructors, and deep copy constructors, define the following function, named reverse, which takes a reference to a Queue as an argument and reverses its contents. After a call to reverse, the Queue’s items are in the reverse order. Note, this is a global function, not a member function of the Queue class.

You may not add member functions to either class. You may only use the interfaces shown on the previous page.

void reverse(Queue &q);

|  |
| --- |
| **void reverse(Queue &q)**  **{**  **// use a stack to reverse the contents of the queue**  **Stack s;**  **// take everything out of q and put it into stack**  **while(!q.empty())**  **{**  **s.push(q.peek());**  **q.pop();**  **}**  **// take everything out of stack and put it back into queue**  **// now it will be reversed**  **while(!s.empty()) {**  **q.push(s.peek());**  **s.pop();**  **}**  **// q is now reversed**  **}**  **// recursion is also acceptable as a solution** |

2. [30 points/3 points each] Read the C++ code below, then evaluate the C++ expressions in the table.

#include <string>

#include <iostream>

#include “List.h” // from lecture (reference code at back of exam)

using std::string;

Class Cup {

protected:

string flavor;

int scoops; // scoops of ice cream

public:

Cup(int s=1, string aflavor=”vanilla”) {

scoops = s;

flavor = aflavor;

}

Cup(string aflavor) {

scoops = 1;

flavor = aflavor;

}

virtual ~Cup(){ std::cout << “Yummy!\n”; }

int getScoops() { return scoops; }

string getFlavor() { return flavor; }

void print() {

std::cout << scoops << “ scoops of “ << flavor << “ ice cream.\n“ ;

}

};

class Concrete: public Cup {

List mix\_ins; // added to ice cream

public:

Concrete(int nscoop, string aflavor=”vanilla”,

string mix1=””, string mix2=””)

{

scoops = nscoop;

flavor = aflavor;

if (mix1 != “”) {

mix\_ins.append(mix1);

if (mix2 != “”) {

mix\_ins.append(mix2);

}

}

}

const List& getMixIns() { return mix\_ins; }

void print() {

std::cout << scoops << “ scoops of “ << flavor << “ ice cream“ ;

List::iterator it = mix\_ins.begin();

if(!it.end()) {

std::cout << “ with”;

while(!it.end()) {

std::cout << “ “ << it.getItem();

it.next();

}

std::cout << “ mixed in.”;

} else {

std::cout << “.”;

}

}

};

int main() {

Cup plain(2);

Concrete \*chocolate = new Concrete(2,”chocolate”,”walnuts”);

Concrete mint = Concrete(3,”vanilla”,”york mint”);

// SNIPPET RUNS HERE

return 0;

}

Evaluate the following expressions as if placed in main at the comment. If the code has a compilation error, explain the error. Otherwise, evaluate the prompt. **It’s possible that any of the code snippets below have an error, even if not prompted to find it. If there is an error, explain what it is.**

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Code snippet** | **Prompt** | **Output, other explanation, or error.** |
| 1 | plain.print(); | What does it print? | **2 scoops of vanilla ice cream.** |
| 2 | cout << mint.getFlavor(); | What does it print? | **Error. Should be std::cout. If it worked, it would print:**  **vanilla** |
| 3 | Concrete small; | What is the problem with this code? | **Error. Concrete doesn’t have a default constructor. Need to at least pass a number of scoops.** |
| 4 | plain.getScoops() + mint.getScoops() | What does this expression evaluate to? | **5** |
| 5 | mint.print(); | What does it print? | **3 scoops of vanilla ice cream with york mint mixed in.** |
| 6 | mint.getMixIns().begin() | What type of object is produced? | **List::iterator** |
| 7 | std::cout << chocolate.getFlavor(); | What does it print? | **Error. Should be chocolate->getFlavor() since chocolate is a pointer.** |
| 8 | Cup kids(“chocolate”);  std::cout << kids.getScoops(); | What does it print? | **1** |
| 9 | delete chocolate; | What does this print?  Hint: virtual destructor. | **Yummy!** |
| 10 | // nothing | What is the output from main if there are no statements added. | **Yummy!**  **Yummy!** |

3. [20 points/5 points each] Give a short code snippet to illustrate the following C++ concepts.

1. Has-A relationship.

|  |
| --- |
| **You can either use private inheritance or just include an object in another object.**  **class A {};**  **class B { A a; };**  **class B has-a A.** |

1. Allocate an object with a specialized constructor on the heap.

|  |
| --- |
| **class A {**  **public:**  **A(int a){} // specialized constructor since it takes an argument. Cannot be default.**  **};**  **int main() {**  **new A(5);**  **return 0;**  **}** |

1. Copy constructor.

|  |
| --- |
| **class A {**  **public:**  **A(const A &copy){} //copy constructo; make a new A from another instance of A**  **};** |

1. Reference.

|  |
| --- |
| **int x;**  **int &y = x; // y references x** |

4. [20 points] Declare a class suitable for representing a singly-linked list. Using that class, implement a function that inserts new nodes at the head of the list. Also, state any assumptions your code makes, such as global variables that have been initialized. No other functions are required, but you may implement other functions if you find it helpful.

|  |
| --- |
| **You can either use the List object I attached or make your own. For the List object, see my notes for a solution, or derive from the following.**  **Here’s a trimmed-down version of what you could have done. Note, I didn’t require that it hold any particular kind of data, so I’m leaving that out of my solution. If you did put data/strings in it, that’s fine, as long as you did so consistently.**  **class List {**  **public: // make it all public for simplicity**  **// class to represent each node of the list**  **class node {**  **public:**  **node\* next;**  **node() { next=nullptr; }**  **};**  **//head pointer**  **node \* head;**  **// REQUIRED: initialize head in List constructor or say that**  **// you will do it in a comment**  **List() { head = nullptr; }**  **// REQUIRED: function to insert at the head**  **void prepend() {**  **node \*new\_node = new node();**  **new\_node->next = head;**  **head = new\_node;**  **}**  **};** |

(left blank for problem 4)

**C++ Keywords**

In common with C:

auto const double float int short struct unsigned  
break continue else for long signed switch void  
case default enum goto register sizeof typedef volatile  
char do extern if return static union while

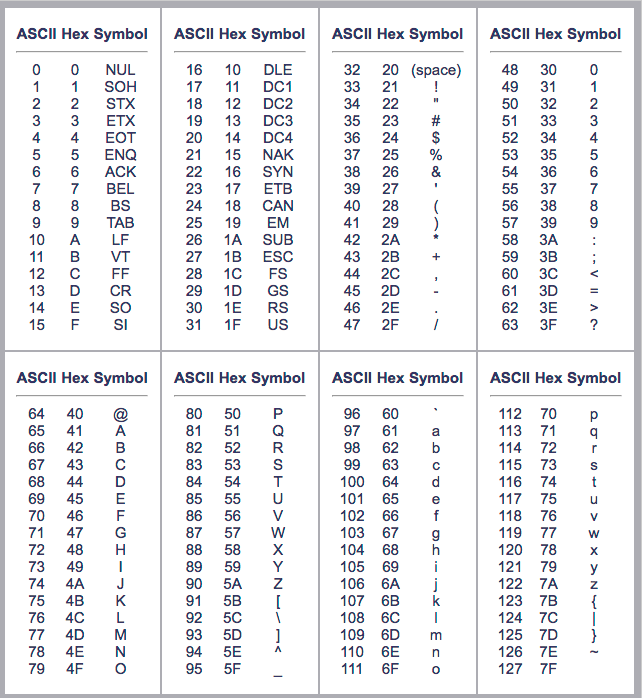
Unique to C++:

asm dynamic\_cast namespace reinterpret\_cast try  
bool explicit new static\_cast typeid  
catch false operator template typename  
class friend private this using  
const\_cast inline public throw virtual  
delete mutable protected true wchar\_t

Reserved words:

and bitand compl not\_eq or\_eq xor\_eq  
and\_eq bitor not or xor

**ASCII Table**



// List.h

#ifndef LIST\_H

#define LIST\_H

#include <string>

class List {

private:

class ListNode {

public:

std::string item;

ListNode \* next;

ListNode(std::string i, ListNode \*n=nullptr);

};

ListNode \* head;

ListNode \* tail;

public:

class iterator {

ListNode \*node;

public:

iterator(ListNode \*n = nullptr);

std::string& getItem();

void next();

bool end();

friend class List;

};

public:

List();

bool empty();

// Only declared, here, implemented

// in List.cpp

void append(std::string a);

bool remove (std::string &copy);

void insertAfter(iterator, std::string);

void removeAfter(iterator, std::string&);

iterator begin() const;

};

#endif