Terence Henriod

Test 1 Corrections

CS 302: Data Structures

October 7, 2013

|  |  |
| --- | --- |
| The Question: | T/F: The expression timePtr->hour is equivalent to \*timePtr.hour. |
|  |  |
| Correct Answer: | False. The second code snippet needs parentheses surrounding the dereferenced pointer before the . operator can be applied appropriately. The . operator has higher precedence than the \* operator, so the code would result in an attempt to access a member of a pointer (which is impossible) and then dereference this member, while in the first snippet the -> operator appropriately dereferences the pointer and selects the hour member all at once. |
|  |  |
| Supporting Quote from Text: | “Operators within the same box have the same precedence, which is higher than operators in lower boxes.”  < The -> and . operators are displayed in a box (table) together, which is above the box that contains the description of the \* operator. > |
|  |  |
| Text and Page Reference: | Data Abstraction and Problem Solving with C++: Walls and Mirrors, by Carrano and Henry; the back of the last page, just as you open the back cover (could be referred to as page 816) |

|  |  |
| --- | --- |
| The Question: | Define the **scope** of a variable. |
|  |  |
| Correct Answer: | The scope of a variable is the context within a program in which it is valid to refer to that variable. This often refers to the context of the implementation code of a function, but I would also argue that this could also refer to the scope of a class. |
|  |  |
| Supporting Quote from Text: | “The scope of a variable is the part of the program where the variable may be used… The first rule of scope you should learn is that a variable cannot be used in any part of the program before the definition.”  “**NOTE:** When a program is running and it enters the section of code that constitutes a variable’s scope, it is said that the variable *comes into scope*. This simply means the variable is now visible and the program may reference it. Likewise, when a variable *leaves scope*, it may no longer be used.”  “When a value is stored in a static member variable, it is not stored in an instance of the class… You can think of static member variables and static member functions as belonging to the class instead of to an instance of the class.” |
|  |  |
| Text and Page Reference: | Starting Out With C++: From Control Structures through Objects, by Gaddis; pages 59, 213, and 800, respectively. (The text for this class does not seem to have a suitable discussion about scope) |

|  |  |
| --- | --- |
| The Question: | The three principles of object-oriented programming are: |
|  |  |
| Correct Answer: | (A) Encapsulation  (B) Inheritance  (C) Polymorphism |
|  |  |
| Supporting Quote from Text: | “  **Note: Three Principles of Object-Oriented Programming**   1. Encapsulation: Objects combine data and operations. 2. Inheritance: Classes can inherit properties from other classes. 3. Polymorphism: Objects can determine appropriate operations at execution time.   ” |
|  |  |
| Text and Page Reference: | Data Abstraction and Problem Solving with C++: Walls and Mirrors, by Carrano and Henry; page 4. |

|  |  |
| --- | --- |
| The Question: | Write the specification file for an node-based list which holds characters. Include all appropriate members.  **Note:** For the test do NOT use compiler guards or templates. |
|  |  |
| Correct Answer: | (Corrections have been underlined)  class CharList  {  public:  CharList();  CharList( const CharList& other );  CharList operator = (const CharList& other );  ~CharList();  bool insert( const char newOne );  bool insertBefore( const char newOne );  bool remove();  void clear();  char getCurs() throw( logic\_error );  bool goToNext();  bool goToPrior();  bool goToBeginning();  bool goToLast();  bool isEmpty() const;  bool isFull() const;  private:  class CharNode  {  public:  CharNode( const char newData, const CharNode\* nextOne );  Char data;  CharNode\* next;  }  CharNode\* head;  CharNode\* cursor;  }; |
|  |  |
| Supporting Quote from Text: | “The assignment operator = requires that we return an object of the same type as rightHandSide, so we must return a LinkedList object. The returned object is placed into the left-hand side of the assignment statement.”  “Passing an argument by reference to a method, especially when the argument is a complex object, saves time and memory, since the method can access or modify the object without copying it.” |
|  |  |
| Text and Page Reference: | Data Abstraction and Problem Solving with C++: Walls and Mirrors, by Carrano and Henry; pages 419 and 35, respectively. |

|  |  |
| --- | --- |
| The Question: | Write the function to insert after the cursor. |
|  |  |
| Correct Answer: | (Corrections have been underlined)  bool CharList::insert( const char newOne )  {  CharNode\* newNode = NULL;  if( isFull() ) // see below for isFull()  {  return false; // for failure  }  else  {  if( head == NULL )  {  newNode = new CharNode( newOne, NULL );  // see below for CharNode implementation  head = newNode;  cursor = head;  }  else  {  newNode = new CharNode( newOne, cursor->next);  cursor->next = newNode;  cursor = newNode;  }  Return true; // for success  }  }  bool CharList::isFull() const  {  // linked list will likely never be full  return false;  }  CharList::CharNode::CharNode( const char newData,  const CharNode\* nextOne )  {  data = newData;  next = nextOne;  } |
|  |  |
| Supporting Quote from Text: | “  bool isFull() const  Returns true if the list is full. Otherwise, returns false.  ” |
|  |  |
| Text and Page Reference: | C++ Data Structures: A Laboratory Course, by Brandle, Geisler, Robergé, and Whittington; page 61. |

|  |  |
| --- | --- |
| The Question: | Write a function that will print **only** the *nth* node in the list (IF there are not n nodes on the list it will print nothing). |
|  |  |
| Correct Answer: | (Corrections have been underlined)  // assumed this function was part of a driver and  // that n-counting starts at 1  // prototype  void printNth( CharList theList, const int nth );  // implementation  void printNth(CharList theList, const int nth )  {  int counter = 1;  if( !isEmpty() ) // isEmpty() implemented below  {  theList.goToBeginning(); // implemented below  while( counter < nth )  {  if( !theList.goToNext() ) // implemented below  {  break;  }  counter ++;  // on the test I incremented the counter before  // advancing the list cursor. this could result in  // undesired behavior if the list were one item  // shorter than n items  }  if( counter == nth )  {  cout << theList.getCurs(); //implemented below  }  }  }  bool CharList::isEmpty() const  {  return ( head == NULL );  }  bool CharList::goToBeginning()  {  if( isEmpty() )  {  return false;  }  else  {  cursor = head;  return true;  }  }  bool CharList::goToNext()  {  if( isEmpty() )  {  return false;  }  else  {  if( cursor->next != NULL )  {  cursor = cursor->next;  return true;  }  else  {  return false;  }  }  }  char CharList::getCurs() throw( logic\_error )  {  if( cursor != NULL )  {  return cursor->data;  }  else  {  throw logic\_error(“Empty”);  }  } |
|  |  |
| Supporting Quote from Text: | “  template<class ItemType>  Node<ItemType>\* LinkedList<ItemType::getNodeAt(  int position ) const  {  // Debugging check of precondition  assert( (position >= 1) && (position <= itemCount) );  // Count from the beginning of the chain  Node<ItemType>\* curPtr = headPtr;  for (int skip = 1; skip < position; skip++)  curPtr = curPtr->getNext();  return curPtr;  // end getNodeAt  }  ” (This function demonstrates that the counter should not be updated until after the cursor is moved, not before like in my mistake on the test) |
|  |  |
| Text and Page Reference: | Data Abstraction and Problem Solving with C++: Walls and Mirrors, by Carrano and Henry; page 275. |