Now write the constructor, as it would appear outside of the class line of code).

Solution:

```
template <class T> Stairs<T>::Stairs(int s, const T& val) {
  size = s;
  data = new T*[s];
  for (int i = 0; i < s; i++) {
    data[i] = new T[i+1];
    for (int j = 0; j <= i; j++) {
      data[i][j] = val;
```

Now write the destructor, as it would appear out of code).

Solution:

```
template <class T> Stairs<T>::~Stairs() {
  for (int i = 0; i < size; i++) {
    delete [] data[i];
  }
  delete [] data;
}
```

```
template <class T>
bool binsearch(const std::vector<T> &v, int low, int high, const T &x) {
 if (high == low) return x == v[low];
  int mid = (low+high) / 2;
  if (x <= v[mid])
    return binsearch(v, low, mid, x);
  else
    return binsearch(v, mid+1, high, x);
}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
  return binsearch(v, 0, v.size()-1, x);
}
```

List functions

List.erase(iterator) iterator points to next position
Return positon of erased iterator

List.insert(iter, n) insert n before iter, iter x change

List.push_back(n)

List.push_front(b)

List.sort(help_function)

List.begin() return first iter; List.end() return last iter

if const list/vector, const iterator

Order Notation

O (log n) dictionary lookup, binary search

O(n log n) sort

Merge sort: space O(n log n) Memory O(n)

const usage:

member function: a() const do not change variable of members, and can only take const member

const int& a: do not change a

const int a: a cannot be changed

const int fun() returned value cannot be changed

const int* fun() returned pointers cannot be changed

cons int fun(cons int & a)const

recursion example

```
Vector functions
v.sort(itr1,itr2,help_function)
v.push_back()
v.front() return reference of first
          change first value;
v.back() return reference of last
         change last value;
```

For each program bug description below, write the letter of the most appropriate debugging skill lem. Each letter should be used at most once.

A) get a backtrace
B) add a breakpoint
C) use step or next
D) add a watchpoint
E) examine different frames of the stack
F) reboot your computer
G) use Dr Memory or Valgrind to locate the le
H) examine variable values in gdb or lldb

tion: E A complex recursive function seems to be entering an infinite loop, despite what I think are perfect base cases.

Solution: G The program always gets the right answer, but when I test it with a complex input dataset that takes a long time to process, my whole computer slows down.

Solution: A I'm unsure where the program is crashing.

Solution: H I've got some tricky math formulas and I suspect I've got an order-of-operations error or a divide-by-zero error.

Solution: D I'm implementing software for a bank, and the value of a customer's bank account is changing in the middle of the month. Interest is only supposed to be added at the end of the month.

```
template <class T>
class Node {
public:
    Node<T>* ptr;
    T value;
};
```

```
bool search_from_loc(loc position, //  current position
                     const vector<string>& board,
                     const string& word,
                     vector<loc>& path)  //  path up to the current pos
{
  // DOUBLE CHECKING OUR LOGIC: the letter at the current board
  // position should equal the next letter in the word
  assert (board[position.row][position.col] == word[path.size()]);

  // start by adding this location to the path
  path.push_back(position);

  // BASE CASE: if the path length matches the word length, we're done!
  if (path.size() == word.size()) return true;

  //  search all the places you can get to in one step
  for (int i = position.row-1; i <= position.row+1; i++) {
    for (int j = position.col-1; j <= position.col+1; j++) {

      // don't walk off the board though!
      if (i < 0 || i >= int(board.size())) continue;
      if (j < 0 || j >= int(board[0].size())) continue;
      // don't consider locations already on our path
      if (on_path(loc(i,j),path)) continue;

      // if this letter matches, recurse!
      if (word[path.size()] == board[i][j]) {
        // if we find the remaining substring, we're done!
        if (search_from_loc (loc(i,j),board,word,path))
          return true;
      }
    }
  }
```

```
int occurrences(const std::vector<std::string> &data, const std::string &element,
                int s1, int s2, int e1, int e2) {
  // s1 & s2 are the current range for the start / first occurence
// e1 & e2 are the current range for the end / last occurence (+1)
  assert (s1 <= s2 && e1 <= e2);
  if (s1 < s2) {
    // first use binary search to find the first occurrence of element
    int mid = (s1 + s2) / 2;
    if (data[mid] >= element)
      return occurrences(data,element,s1,mid,e1,e2);
    return occurrences(data,element,mid+1,s2,e1,e2);
  } else if (e1 < e2) {
    // then use binary search to find the last occurrence of element (+1)
    int mid = (e1 + e2) / 2;
    if (data[mid] > element)
      return occurrences(data,element,s1,s2,e1,mid);
    return occurrences(data,element,s1,s2,mid+1,e2);
  } else {
    // the simply subtract these indices
    assert (s1 == s2 && e1 == e2 && e1 >= s1);
    return e1 - s1;
  }
}
```

```
int occurrences(const std::vector<std::string> &data, const std::string &element) {
  // use binary seach twice to find the first & last occurrence of element
  return occurrences(data,element,0,data.size(),0,data.size());
}
```

Solution: B) Once you've found the general area of the problem, it can be helpful to add a breakpoint shortly before the crash, so you can examine the situation more closely. C) Once you've decided the state of the program is reasonable, you can advance the program one line at a time using next or step into a helper function that may be causing problems. Rebooting your computer is unlikely to fix a bug in your own code.

A) use of uninitialized memory
B) mismatched new/delete/delete[]
C) memory leak
D) already freed memory
E) no memory error
F) invalid write

Solution: B

```
char* a = new char[6];
a[0] = 'B';  a[1] = 'y';
a[2] = 'e';  a[3] = '\0';
cout << a << endl;
delete a;
```

Solution: A

```
int a[10];
int b[5];
for(int i=10; i>5; i--){
  a[((i-6)*2+1)] = i*2;
  a[((i-6)*2)] = b[i-6];
  cout << a[(i-6)*2] << endl;
}
```

Solution: F

```
bool* is_even = new bool[10];
for(int i=0; i<=10; i++){
  is_even[i] = ((i%2)==0);
}
delete [] is_even;
```

Solution: C

```
int a[2];
float** b = new float*[2];
b[0] = new float[1];
a[0] = 5; a[1] = 2;
b[0][0] = a[0]*a[1];
delete [] b[0];
b[0] = new float[0];
delete [] b;
```

Solution: D

```
string* str1 = new string;
string* str2;
string* str3 = new string;
*str1 = "Hello";
str2 = str1;
*str3 = *str1;
delete str1;
delete str3;
delete str2;
```

Solution: E

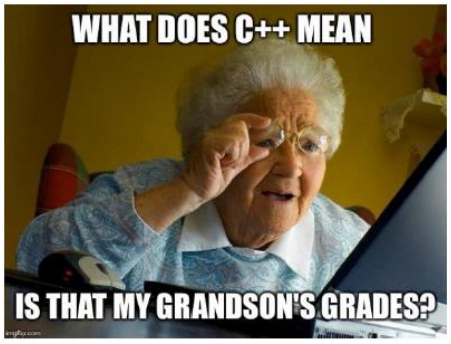
```
int x[3];
int* y = new int[3];
for (int i=3; i>=1; i--){
  y[i-1] = i*i;
  x[i-1] = y[i-1]*y[i-1];
}
delete [] y;
```

```
template <class T> class Node {
public:
  // Default constructor
  Node() : prev(NULL), num_elements(0), next(NULL) {}
  // Copy constructor
  Node(const Node<T> &old)
      : prev(NULL), num_elements(old.num_elements), next(NULL) {
    for (int i = 0; i < old.num_elements; i++) {
      elements[i] = old.elements[i];
    }
  }
  Node<T> *prev;
  int num_elements;
  T elements[NUM_ELEMENTS_PER_NODE];
  Node<T> *next;
};
```

Any iterators attached to an STL vector should be assumed to be invalid after a call to push back (or erase or resize) because the internal dynamically allocated array may have been relocated in memory (or the data shifted). Dereferencing the pre-push back iterators to print the data is dangerous since that memory may have been deleted/freed.

What is the purpose of typename: Sometimes when using templates you need to add it unconfused the compiler.
Which of the following is about return by reference: if memory is abundant
Which of the following is not true about memory debugger
A memory debugger points to the line number in the code that must.
Which of the following statements about sol vectors and sol lists is not true

```
// default constructor, copy constructor, assignment operator, destructor.
UnrolledLL() : head(NULL), tail(NULL), size_(0) {}
UnrolledLL(const UnrolledLL<T> &old) { this->copy_list(old); }
UnrolledLL &operator=(const UnrolledLL<T> &old);
~UnrolledLL() { this->destroy_list(); }
```




```
void merge(int low, int mid, int high, vector<T>& values, vector<T>& scratch) {

  // some output so we can watch how merge sort works
  cout << "merge:  low = " << low << ", mid = " << mid << ", high = " << high << endl;
  int i=low, j=mid+1, k=low;
  // int p;
  /*
  cout << "LOW INTERVAL:     ";
  for (int p = low; p <= mid; p++)
    cout << values[p] << " ";
  cout << endl << "HIGH INTERVAL:    ";
  for (int p = mid+1; p <= high; p++)
    cout << values[p] << " ";
  cout << endl;
  */

  // while there's still something left in one of the sorted subintervals...
  while (i <= mid && j <= high) {

    // look at the top values, grab the smaller one, store it in the scratch vector
    if (values[i] < values[j]) {
      scratch[k] = values[i]; ++i;
    } else {
      scratch[k] = values[j]; ++j;
    }
    ++k;
  }

  // Copy the remainder of the interval that hasn't been exhausted
  // Note: only one of for loops will do anything (have a non-zero index range)
  for ( ; i<=mid; ++i, ++k ) scratch[k] = values[i];  // low interval
  for ( ; j<=high; ++j, ++k ) scratch[k] = values[j]; // high interval

  // Copy from scratch back to values
  for ( i=low; i<=high; ++i ) values[i] = scratch[i];
```

```
template <class T> void RemoveAll(Node<T>*& head) {
  if(head){
    //If there's something to remove
    Node<T>* dummy = head; //Save a copy of current head
    head = head->next; //Make head point to the next thing
    delete dummy;
    RemoveAll(head);
  }
}
```