CSCI-1200 Data Structures — Fall 2014
Lecture 7 — Templated Classes & Vector Implementation

Review from Lectures 5 & 6

- Arrays and pointers
- Different types of memory ("automatic", static, dynamic)
- Dynamic allocation of arrays
- Drawing pictures to explain stack vs. heap memory allocation.

7.1 Today's Lecture

- Designing our own container classes:
  - Mimic the interface of standard library (STL) containers
  - Study the design of memory management.
  - Move toward eventually designing our own, more sophisticated classes.
- Vector implementation
- Templated classes (*including* compilation and instantiation of templated classes)
- Copy constructors, assignment operators, and destructors
- Memory Debuggers

Optional Reading: Ford&Topp, Sections 5.3-5.5; Koening & Moo Chapter 11

7.2 Vector Public Interface

- In creating our own version of the STL vector class, we will start by considering the public interface:

```
public:
  // MEMBER FUNCTIONS AND OTHER OPERATORS
  T& operator[] (size_type i);
  const T& operator[] (size_type i) const;
  void push_back(const T& t);
  void resize(size_type n, const T& fill_in_value = T());
  void clear();
  bool empty() const;
  size_type size() const;
```

- To implement our own generic (a.k.a. templated) vector class, we will implement all of these operations, manipulate the underlying representation, and discuss memory management.

7.3 Templated Class Declarations and Member Function Definitions

- In terms of the layout of the code in `vec.h` (pages 5 & 6 of the handout), the biggest difference is that this is a *templated class*. The keyword `template` and the template type name must appear before the class declaration:

```
template <class T> class Vec
```

- Within the class declaration, `T` is used as a type and all member functions are said to be "templated over type `T`". In the actual text of the code files, templated member functions are often defined (written) *inside the class declaration*.
- The templated functions defined outside the template class declaration must be preceded by the phrase: `template <class T>` and then when `Vec` is referred to it must be as `Vec<T>` . For example, for member function `create` (two versions), we write:

```
template <class T> void Vec<T>::create(...
```

7.4 Syntax and Compilation

- Templated classes and templated member functions are not created/compiled/instantiated until they are needed. Compilation of the class declaration is triggered by a line of the form: `Vec<int> v1;` with `int` replacing `T`. This also compiles the default constructor for `Vec<int>` because it is used here. Other member functions are not compiled unless they are used.
- When a different type is used with `Vec`, for example in the declaration: `Vec<double> z;` the template class declaration is compiled again, this time with `double` replacing `T` instead of `int`. Again, however, only the member functions used are compiled.
- This is very different from ordinary classes, which are usually compiled separately and all functions are compiled regardless of whether or not they are needed.
- The templated class declaration and the code for all used member functions must be provided where they are used. As a result, member functions definitions are often included within the class declaration or defined outside of the class declaration but still in the `.h` file. If member function definitions are placed in a separate `.cpp` file, this file must be `#include`-d, just like the `.h` file, because the compiler needs to see it in order to generate code. (Normally we don't `#include .cpp` files!) See also diagram on page 7 of this handout.

  Note: Including function definitions in the `.h` for ordinary non-templated classes may lead to compilation errors about functions being "multiply defined". Some of you have already seen these errors.

7.5 Member Variables

Now, looking inside the `Vec<T>` class at the member variables:

- `m_data` is a pointer to the start of the array (after it has been allocated). Recall the close relationship between pointers and arrays.
- `m_size` indicates the number of locations currently in use in the vector. This is exactly what the `size()` member function should return,
- `m_alloc` is the total number of slots in the dynamically allocated block of memory.

Drawing pictures, which we will do in class, will help clarify this, especially the distinction between `m_size` and `m_alloc`.

7.6 Typedefs

- Several types are created through `typedef` statements in the first `public` area of `Vec`. Once created the names are used as ordinary type names. For example `Vec<int>::size_type` is the return type of the size() function, defined here as an `unsigned int`.

7.7 operator[]

- Access to the individual locations of a `Vec` is provided through `operator[]`. Syntactically, use of this operator is translated by the compiler into a call to a function called `operator[]`. For example, if `v` is a `Vec<int>`, then:

  ```
  v[i] = 5;
  ```

  translates into:

  ```
  v.operator[](i) = 5;
  ```

- In most classes there are two versions of `operator[]`:
  - A non-const version returns a reference to `m_data[i]`. This is applied to non-const `Vec` objects.
  - A const version is the one called for `const Vec` objects. This also returns `m_data[i]`, but as a const reference, so it can not be modified.

7.8 Default Versions of Assignment Operator and Copy Constructor Are Dangerous!

- Before we write the copy constructor and the assignment operator, we consider what would happen if we didn't write them.
- C++ compilers provide default versions of these if they are not provided. These defaults just copy the values of the member variables, one-by-one. For example, the default copy constructor would look like this:

```
template <class T>
Vec<T> :: Vec(const Vec<T>& v)
 : m_data(v.m_data), m_size(v.m_size), m_alloc(v.m_alloc)
{}
```

  In other words, it would construct each member variable from the corresponding member variable of `v`. This is dangerous and incorrect behavior for the `Vec` class. We don't want to just copy the `m_data` pointer. We really want to create a copy of the entire array! Let's look at this more closely...

7.9 Exercise

Suppose we used the default version of the assignment operator and copy constructor in our `Vec<T>` class. What would be the output of the following program? Assume all of the operations **except** the copy constructor behave as they would with a `std::vector<double>`.

```
Vec<double> v(4, 0.0);
v[0] = 13.1;  v[2] = 3.14;
Vec<double> u(v);
u[2] = 6.5;
u[3] = -4.8;
for (unsigned int i=0; i<4; ++i)
  cout << u[i] << " " << v[i] << endl;
```

Explain why this happens by drawing a picture of the memory of both `u` and `v`.

7.10 Classes With Dynamically Allocated Memory

- For `Vec` (and other classes with dynamically-allocated memory) to work correctly, each object must do its own dynamic memory allocation and deallocation. We must be careful to keep the memory of each object instance separate from all others.
- All dynamically-allocated memory for an object should be released when the object is finished with it or when the object itself goes out of scope (through what's called a *destructor*).
- To prevent the creation and use of default versions of these operations, we must write our own:
  - *Copy constructor*
  - *Assignment operator*
  - *Destructor*

7.11 Copy Constructor

- This constructor must dynamically allocate any memory needed for the object being constructed, copy the contents of the memory of the passed object to this new memory, and set the values of the various member variables appropriately.
- **Exercise:** In our `Vec` class, the actual copying is done in a private member function called `copy`. Write the private member function `copy`.

7.12 The "this" pointer

- All class objects have a special pointer defined called `this` which simply points to the current class object, and it may not be changed.
- The expression `*this` is a reference to the class object.
- The `this` pointer is used in several ways:
  - Make it clear when member variables of the current object are being used.
  - Check to see when an assignment is self-referencing.
  - Return a reference to the current object.

7.13 Assignment Operator

- Assignment operators of the form: `v1 = v2;`
  are translated by the compiler as: `v1.operator=(v2);`
- Cascaded assignment operators of the form: `v1 = v2 = v3;`
  are translated by the compiler as: `v1.operator=(v2.operator=(v3));`
- Therefore, the value of the assignment operator (`v2 = v3`) must be suitable for input to a second assignment operator. This in turn means the result of an assignment operator ought to be a reference to an object.
- The implementation of an assignment operator usually takes on the same form for every class:
  - Do no real work if there is a self-assignment.
  - Otherwise, destroy the contents of the current object then copy the passed object, just as done by the copy constructor. In fact, it often makes sense to write a private helper function used by both the copy constructor and the assignment operator.
  - Return a reference to the (copied) current object, using the `this` pointer.

7.14 Destructor (the "constructor with a tilde/twiddle")

- The destructor is called implicitly when an automatically-allocated object goes *out of scope* or a dynamically-allocated object is *deleted*. It can never be called explicitly!
- The destructor is responsible for deleting the dynamic memory "owned" by the class.
- The syntax of the function definition is a bit weird. The `~` has been used as a logic negation in other contexts.

7.15 Increasing the Size of the Vec

- `push_back(T const& x)` adds to the end of the array, increasing `m_size` by one `T` location. But what if the allocated array is full (`m_size == m_alloc`)?
  1. Allocate a new, larger array. The best strategy is generally to double the size of the current array. Why?
  2. If the array size was originally 0, doubling does nothing. We must be sure that the resulting size is at least 1.
  3. Then we need to copy the contents of the current array.
  4. Finally, we must delete current array, make the `m_data` pointer point to the start of the new array, and adjust the `m_size` and `m_alloc` variables appropriately.
- Only when we are sure there is enough room in the array should we actually add the new object to the back of the array.

7.16 Exercises

- Finish the definition of `Vec::push_back`.
- Write the `Vec::resize` function.

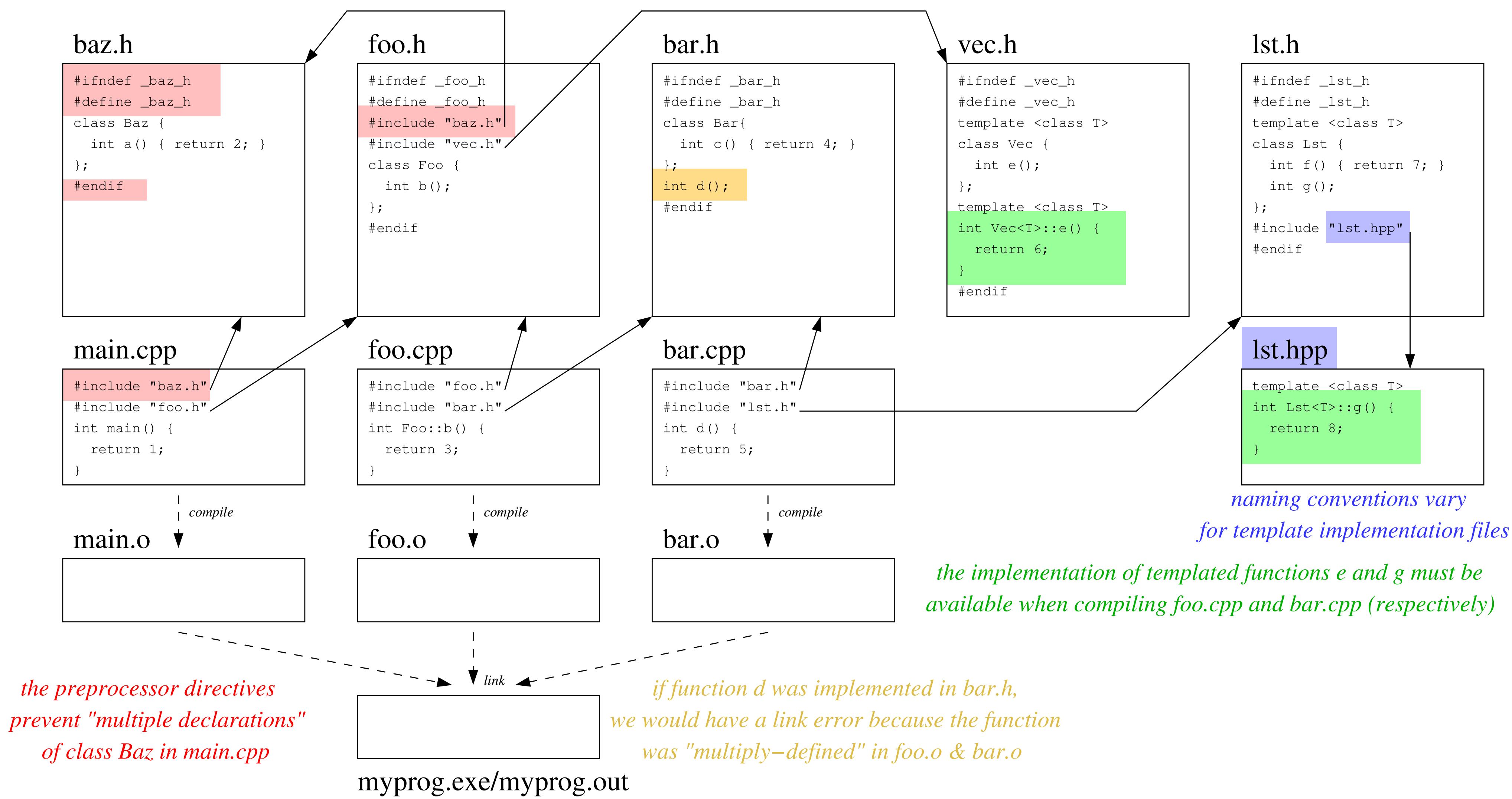
7.17 Vec Declaration & Implementation (`vec.h`)

```
#ifndef Vec_h_
#define Vec_h_
// Simple implementation of the vector class, revised from Koenig and Moo.  This
// class is implemented using a dynamically allocated array (of templated type T).
// We ensure that that m_size is always <= m_alloc and when a push_back or resize
// call would violate this condition, the data is copied to a larger array.

template <class T> class Vec {

public:
  // TYPEDEFS
  typedef unsigned int size_type;

  // CONSTRUCTORS, ASSIGNMNENT OPERATOR, & DESTRUCTOR
  Vec() { this->create(); }
  Vec(size_type n, const T& t = T()) { this->create(n, t); }
  Vec(const Vec& v) { copy(v); }
  Vec& operator=(const Vec& v);
  ~Vec() { delete [] m_data; }

  // MEMBER FUNCTIONS AND OTHER OPERATORS
  T& operator[] (size_type i) { return m_data[i]; }
  const T& operator[] (size_type i) const { return m_data[i]; }
  void push_back(const T& t);
  void resize(size_type n, const T& fill_in_value = T());
  void clear() { delete [] m_data;  create(); }
  bool empty() const { return m_size == 0; }
  size_type size() const { return m_size; }

private:
  // PRIVATE MEMBER FUNCTIONS
  void create();
  void create(size_type n, const T& val);
  void copy(const Vec<T>& v);

  // REPRESENTATION
  T* m_data;         // Pointer to first location in the allocated array
  size_type m_size;  // Number of elements stored in the vector
  size_type m_alloc; // Number of array locations allocated,  m_size <= m_alloc
};

// Create an empty vector (null pointers everywhere).
template <class T>  void Vec<T>::create() {
  m_data = NULL;
  m_size = m_alloc = 0;  // No memory allocated yet
}

// Create a vector with size n, each location having the given value
template <class T> void Vec<T>::create(size_type n, const T& val) {
  m_data = new T[n];
  m_size = m_alloc = n;
  for (size_type i = 0; i < m_size; i++) {
    m_data[i] = val;
  }
}

// Assign one vector to another, avoiding duplicate copying.
template <class T> Vec<T>& Vec<T>::operator=(const Vec<T>& v) {
  if (this != &v) {
    delete [] m_data;
    this -> copy(v);
  }
  return *this;
}
```

```
// Create the vector as a copy of the given vector.
template <class T> void Vec<T>::copy(const Vec<T>& v) {









}

// Add an element to the end, resize if necesssary.
template <class T> void Vec<T>::push_back(const T& val) {
  if (m_size == m_alloc) {
    // Allocate a larger array, and copy the old values











  }
  // Add the value at the last location and increment the bound
  m_data[m_size] = val;
  ++ m_size;
}

// If n is less than or equal to the current size, just change the size.  If n is
// greater than the current size, the new slots must be filled in with the given value.
// Re-allocation should occur only if necessary.  push_back should not be used.
template <class T> void Vec<T>::resize(size_type n, const T& fill_in_value) {
















}

#endif
```

7.18 File Organization & Compilation of Templated Classes

The diagram on the next page shows the typical and suggested file organization for non-templated vs. templated classes. Common mistakes and the resulting compilation errors are noted.



baz.h
#ifndef _baz_h
#define _baz_h
class Baz {
int a() { return 2; }
};
#endif
foo.h
#ifndef _foo_h
#define _foo_h
#include "baz.h"
#include "vec.h"
class Foo {
int b();
};
#endif
bar.h
#ifndef _bar_h
#define _bar_h
class Bar{
int c() { return 4; }
};
int d();
#endif
vec.h
#ifndef _vec_h
#define _vec_h
template <class T>
class Vec {
int e();
};
template <class T>
int Vec<T>::e() {
return 6;
}
#endif
lst.h
#ifndef _lst_h
#define _lst_h
template <class T>
class Lst {
int f() { return 7; }
int g();
};
#include "lst.hpp"
#endif
main.cpp
#include "baz.h"
#include "foo.h"
int main() {
return 1;
}
foo.cpp
#include "foo.h"
#include "bar.h"
int Foo::b() {
return 3;
}
bar.cpp
#include "bar.h"
#include "lst.h"
int d() {
return 5;
}
lst.hpp
template <class T>
int Lst<T>::g() {
return 8;
}
compile
compile
compile
main.o
foo.o
bar.o
naming conventions vary
for template implementation files
the implementation of templated functions e and g must be
available when compiling foo.cpp and bar.cpp (respectively)
link
the preprocessor directives
prevent "multiple declarations"
of class Baz in main.cpp
if function d was implemented in bar.h,
we would have a link error because the function
was "multiply−defined" in foo.o & bar.o
myprog.exe/myprog.out

7.19 Memory Debugging

In addition to the step-by-step debuggers like `gdb`, `lldb`, or the debugger in your IDE, we recommend using a memory debugger like “Dr. Memory” (Windows, Linux, and MacOSX) or “Valgrind” (Linux and MacOSX). These tools can detect the following problems:

- Use of uninitialized memory
- Reading/writing memory after it has been free’d
- Reading/writing off the end of malloc’d blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks - where pointers to malloc’d blocks are lost forever
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions

7.20 Sample Buggy Program

Can you see the errors in this program?

```
 1 #include <iostream>
 2
 3 int main() {
 4
 5   int *p = new int;
 6   if (*p != 10) std::cout << "hi" << std::endl;
 7
 8   int *a = new int[3];
 9   a[3] = 12;
10   delete a;
11
12 }
```

7.21 Using Dr. Memory `http://www.drmemory.org`

Here’s how Dr. Memory reports the errors in the above program:

```
~~Dr.M~~ Dr. Memory version 1.8.0
~~Dr.M~~
~~Dr.M~~ Error #1: UNINITIALIZED READ: reading 4 byte(s)
~~Dr.M~~ # 0 main               [memory_debugger_test.cpp:6]
hi
~~Dr.M~~
~~Dr.M~~ Error #2: UNADDRESSABLE ACCESS beyond heap bounds: writing 4 byte(s)
~~Dr.M~~ # 0 main               [memory_debugger_test.cpp:9]
~~Dr.M~~ Note: refers to 0 byte(s) beyond last valid byte in prior malloc
~~Dr.M~~
~~Dr.M~~ Error #3: INVALID HEAP ARGUMENT: allocated with operator new[], freed with operator delete
~~Dr.M~~ # 0 replace_operator_delete              [/drmemory_package/common/alloc_replace.c:2684]
~~Dr.M~~ # 1 main                                 [memory_debugger_test.cpp:10]
~~Dr.M~~ Note: memory was allocated here:
~~Dr.M~~ Note: # 0 replace_operator_new_array              [/drmemory_package/common/alloc_replace.c:2638]
~~Dr.M~~ Note: # 1 main                                    [memory_debugger_test.cpp:8]
~~Dr.M~~
~~Dr.M~~ Error #4: LEAK 4 bytes
~~Dr.M~~ # 0 replace_operator_new              [/drmemory_package/common/alloc_replace.c:2609]
~~Dr.M~~ # 1 main                              [memory_debugger_test.cpp:5]
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       1 unique,     1 total unaddressable access(es)
~~Dr.M~~       1 unique,     1 total uninitialized access(es)
```

```
~~Dr.M~~       1 unique,     1 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       1 unique,     1 total,      4 byte(s) of leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of possible leak(s)
~~Dr.M~~ Details: /DrMemory-MacOS-1.8.0-8/drmemory/logs/DrMemory-a.out.7726.000/results.txt
```

And the fixed version:

```
~~Dr.M~~ Dr. Memory version 1.8.0
hi
~~Dr.M~~
~~Dr.M~~ NO ERRORS FOUND:
~~Dr.M~~       0 unique,     0 total unaddressable access(es)
~~Dr.M~~       0 unique,     0 total uninitialized access(es)
~~Dr.M~~       0 unique,     0 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of possible leak(s)
~~Dr.M~~ Details: /DrMemory-MacOS-1.8.0-8/drmemory/logs/DrMemory-a.out.7762.000/results.txt
```

Note: Dr. Memory on Windows with the Visual Studio compiler may not report a mismatched free() / delete / delete [] error (e.g., line 10 of the sample code above). This may happen if optimizations are enabled and the objects stored in the array are simple and do not have their own dynamically-allocated memory that lead to their own indirect memory leaks.

7.22 Using Valgrind `http://valgrind.org/`

And this is how Valgrind reports the same errors:

```
==31226== Memcheck, a memory error detector
==31226== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==31226== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==31226== Command: ./a.out
==31226==
==31226== Conditional jump or move depends on uninitialised value(s)
==31226==    at 0x40096F: main (memory_debugger_test.cpp:6)
==31226==
hi
==31226== Invalid write of size 4
==31226==    at 0x4009A3: main (memory_debugger_test.cpp:9)
==31226==  Address 0x4c3f09c is 0 bytes after a block of size 12 alloc'd
==31226==    at 0x4A0700A: operator new[](unsigned long) (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==31226==    by 0x400996: main (memory_debugger_test.cpp:8)
==31226==
==31226== Mismatched free() / delete / delete []
==31226==    at 0x4A07991: operator delete(void*) (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==31226==    by 0x4009B4: main (memory_debugger_test.cpp:10)
==31226==  Address 0x4c3f090 is 0 bytes inside a block of size 12 alloc'd
==31226==    at 0x4A0700A: operator new[](unsigned long) (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==31226==    by 0x400996: main (memory_debugger_test.cpp:8)
==31226==
==31226==
==31226== HEAP SUMMARY:
==31226==     in use at exit: 4 bytes in 1 blocks
==31226==   total heap usage: 2 allocs, 1 frees, 16 bytes allocated
==31226==
==31226== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==31226==    at 0x4A06965: operator new(unsigned long) (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==31226==    by 0x400961: main (memory_debugger_test.cpp:5)
==31226==
==31226== LEAK SUMMARY:
==31226==    definitely lost: 4 bytes in 1 blocks
```

```
==31226==    indirectly lost: 0 bytes in 0 blocks
==31226==      possibly lost: 0 bytes in 0 blocks
==31226==    still reachable: 0 bytes in 0 blocks
==31226==         suppressed: 0 bytes in 0 blocks
==31226==
==31226== For counts of detected and suppressed errors, rerun with: -v
==31226== Use --track-origins=yes to see where uninitialised values come from
==31226== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 2 from 2)
```

And here's what it looks like after fixing those bugs:

```
==31252== Memcheck, a memory error detector
==31252== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==31252== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==31252== Command: ./a.out
==31252==
hi
==31252==
==31252== HEAP SUMMARY:
==31252==     in use at exit: 0 bytes in 0 blocks
==31252==   total heap usage: 2 allocs, 2 frees, 16 bytes allocated
==31252==
==31252== All heap blocks were freed -- no leaks are possible
==31252==
==31252== For counts of detected and suppressed errors, rerun with: -v
==31252== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

7.23 How to use a memory debugger

- Detailed instructions on installation & use of these tools are available here:
  http://www.cs.rpi.edu/academics/courses/fall14/csci1200/memory_debugging.php
- **Memory errors** (uninitialized memory, out-of-bounds read/write, use after free) may cause seg faults, crashes, or strange output.
- **Memory leaks** on the other hand will never cause incorrect output, but your program will be inefficient and hog system resources. A program with a memory leak may waste so much memory it causes all programs on the system to slow down significantly or it may crash the program or the whole operating system if the system runs out of memory (this takes a while on modern computers with lots of RAM & harddrive space).
- For HW3, the homework submission server is configured to run your code with Dr. Memory to search for memory problems and present the output with the submission results. For full credit your program must be memory error and memory leak free!
- A program that seems to run perfectly fine on one computer may still have significant memory errors. Running a memory debugger will help find issues that might break your homework on another computer or when submitted to the homework server.
- **Important Note:** When these tool find a memory leak, they point to the line of code where this memory was *allocated*. These tools does not understand the program logic and thus obviously cannot tell us where it *should* have been deleted.
- A final note: STL and other 3rd party libraries are highly optimized and sometimes do sneaky but correct and bug-free tricks for efficiency that confuse the memory debugger. For example, because the STL string class uses its own allocator, there may be a warning about memory that is "still reachable" even though you've deleted all your dynamically allocated memory. The memory debuggers have automatic suppressions for some of these known "false positives", so you will see this listed as a "suppressed leak". So don't worry if you see those messages.