Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```bash
cd cs301
mkdir debugging
cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):

```bash
cp ~lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

- CHTNode.cpp
- CHTNode.h
- ChainedHashTable.cpp
- ChainedHashTable.h
- DuplicateKeyException.h
- KeyNotFoundException.h
- Makefile
- tester.cpp

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You could compile the test program by hand with the following command:

```bash
g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would
end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean
that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize
this by allowing separate compilation of multiple files to create “object modules” (for
C++, these files end in a “.o” extension) which can be linked together to form an execut-
able in a separate stage of compilation. So, you could compile all of the files separately,
using the “-c” flag, then link them together at the end:

```bash
g++ -g -c tester.cpp
 g++ -g -c CHTNode.cpp
 g++ -g -c ChainedHashTable.cpp
 g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the
final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For
example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it
changes, both of the .cpp files should be recompiled. Unix provides the make utility to
automate this process. The make utility takes a file called “Makefile” as input and auto-
matically figures out which files need to be recompiled based on the rules that Makefile
contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging direc-
tory. It consists of a sequence of “targets” followed by commands to be executed to create
the targets. A line like:

```bash
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h \
           CHTNode.h
```
indicates that the “target” ChainedHashTable.o should be recompiled if any of the three
files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified
since ChainedHashTable.o was created. (The “\” is required to indicate that the depen-
dency list spans more than one line.) The following line in the Makefile is the command to
execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if
you have occasion to modify a Makefile or create one of your own, note that the first char-
acter of these command lines MUST be a tab, otherwise you’ll get very strange error mes-
sages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in
the directory containing the files. When it finishes, type “ls” again. You should now see
“.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable
we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type
   ```bash
   ddd <executable-name>
   ```
   where `<executable-name>` is replaced with the name of the executable you want to
debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

Setting a breakpoint

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

Examining variable values

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

   t.table[0]@t.size

into the Argument window (next to the “( ):” label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. **Double-click on the 0th element of the table array.** A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. **Use the mouse to drag this box up and away from the table display box.** You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. **Double-click on the key value.** This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. **Double click on the _M_dataplus field of key._M_p** in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

**Stepping through a program**

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variables. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert(“seventh”, 7) to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump( )” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the formal parameters have been give for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

**Debugging a “core dump”**

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ run time system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type
   
   ./tester

   You should see some output from the program followed by the message
   
   Segmentation Fault (core dumped)

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type
   
   ddd tester core &

   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name mangling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “_ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHTNode. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

**Exercise**

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
**ddd Quick Reference**

**Running ddd**
ddd `<executable-name>` where the name of your executable is substituted for `<executable-name>`

**Viewing program output/typing console input**
Choose View->Execution window

**Setting a breakpoint**
Click to the left of the line where you want to stop, then click the “Break” button.

**Running the program**
Click “Run” in the Command tool or choose Program->Run

**Viewing variable values**
“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

**Viewing the contents of dynamically allocated arrays**
Use “@” to specify the length of the array in the Argument area (e.g. a[0]@7)

**Following pointers**
Double-clicking on a pointer displays what it points to in the Data window.

**Single stepping**
“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.
“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

**Executing to the end of a function/method**
To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

**Finding the source of a core dump**
Type “ddd `<executable> core” then choose Status->Backtrace.”
#include <string>

class CHTNode {
private:
    std::string key;
    int data;

public:
    CHTNode *next;

    CHTNode(std::string k, int payload);
    std::string getKey();
    int getData();
};

// Constructor
CHTNode::CHTNode(std::string k, int payload) {
    key = k;
    data = payload;
}

// Return the key value stored in this node
std::string CHTNode::getKey() {
    return key;
}

// Return the data value stored in this node.
int CHTNode::getData() {
    return data;
}
A templated Chained Hash table. The keys are assumed to be strings. Duplicate keys are not allowed.

// File: ChainedHashTable.h
// Author: Lewis Barnett

// The "declaration" of the class, containing declaration of data members
// and prototypes for methods.
class ChainedHashTable {
    public:
        ChainedHashTable(int slots);
        void insert (std::string k, int value);
        bool lookup (std::string k);
        int retrieve (std::string k);
        void remove(std::string k);
        void dump();

    private:
        // Declaration for a dynamically allocated array of pointers to
        // CHTNodes -- this is the "array of linked lists" that serves
        // as our hash table.
        CHTNode **table;
        // size is the number of slots in table. C++ arrays don’t remember
        // this after they’re created, so we have to save it ourselves.
        int size;

        // Private helper method that generates the hash value for a
        // given key.
        int hash (std::string k);
};
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include "ChainedHashTable.h"

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
  // This says: create an array of pointers to CHTNodes with
  // “slots” elements.
  table = new CHTNode* [slots];
  size = slots;
  for (int i = 0; i < size; i++) {
    table[i] = 0;// Make each element of table a null pointer.
  }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
  if (lookup(k)) {
    // Note that we don’t have to create the exception object with
    // new as we did in Java.
    throw DuplicateKeyException(k);
  }

  int bucket = hash(k);
  CHTNode *n = new CHTNode(k, value);

  // Insert the new (key, value) pair at the beginning of the chain
  // for this bucket.
  n->next = table[bucket];
  table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
  bool found = false;
  int bucket = hash(k);

  CHTNode *p = table[bucket];
while (!found && p != 0) {
  if (k == p->getKey())
    found = true;
  p = p->next;
}

return found;

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
  bool found = false;
  int bucket = hash(k);

  CHTNode *p = table[bucket];
  while (!found) {
    if (k == p->getKey())
      found = true;
    p = p->next;
  }

  if (found) {
    return p->getData();
  } else {
    throw KeyNotFoundException(k);
  }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
  // First, find the position of the requested key
  bool found = false;
  int bucket = hash(k);

  // Scan the list for the correct bucket to find the key
  CHTNode *p = table[bucket];
  while (!found && p != 0) {
    if (k == p->getKey()) {
      found = true;
      break;
    }
    p = p->next;
  }
// Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;

        prev->next = p->next;

        // This is necessary because C++ doesn’t automatically
        // garbage collect objects when there are no longer any
        // references to them.
        delete p;
    }
} else {
    throw KeyNotFoundException(k);
}
}

// This is essentially identical to program 14.1 in Sedgewick’s Algo-
// rithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
    int h = 0;
    int a = 127;

    for (int i = 0; i < k.length(); i++){
        h = (a * h + k[i]) % size;
    }

    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump(){
    std::cout << "Contents of table:" << std::endl << std::endl;

    for (int i = 0; i < size; i++) {
        std::cout << "Bucket " << i << " : " << std::endl;
        CHTNode *p = table[i];
while (p != 0) {
    std::cout << "\tkey = " << p->getKey() << " value = " << p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
// tester.cpp
#include <iostream>
#include <string>
#include "CHTNode.h"
#include "ChainedHashTable.h"

main() {
    // Declare a new hash table that stores ints as its data value.
    ChainedHashTable t(11);

    // Insert a number of values. We end up with some chains of
    // length 2.
    t.insert("first", 1);
    t.insert("second", 2);
    t.insert("frist", 32);
    t.insert("third", 3);
    t.insert("fourth", 4);
    t.insert("fifth", 5);
    t.insert("sixth", 6);
    t.insert("seventh", 7);

    // Test the response to inserting a duplicate value.
    try {
        t.insert("third", 9);
    } catch (DuplicateKeyException e) {
        // std::cerr is a separate output stream used for error reporting.
        // Even if the standard output stream is redirected to a file,
        // std::cerr's output appears on the screen.
        std::cerr << "Attempted to insert duplicate key " << e.key
                << std::endl;
    }

    t.dump();

    int val = t.retrieve("sixth");
    std::cout << "retrieve("sixth") returns " << val << std::endl;

    try {
        t.retrieve("bungle");
    } catch (KeyNotFoundException e) {
        std::cerr << "Didn't find key " << e.key << " in retrieve." <<
                 std::endl << std::endl;
    }

    t.remove("sixth");

    std::cout << "After removing ""sixth"":" << std::endl;
t.dump();

try {
    t.remove("junk");
} catch (KeyNotFoundException e) {
    std::cerr << "Didn’t find key “ << e.key << “ in remove.” <<
              std::endl << std::endl;
}

    t.remove("fourth");
    std::cout << “After removing "fourth":” << std::endl;
    t.dump();

    t.remove("seventh");
    std::cout << “After removing "seventh":” << std::endl;
    t.dump();
}
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```bash
cd cs301
mkdir debugging
cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):  
```bash
cp ~lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

- CHTNode.cpp
- CHTNode.h
- ChainedHashTable.cpp
- ChainedHashTable.h
- DuplicateKeyException.h
- KeyNotFoundException.h
- Makefile
- tester.cpp

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You *could* compile the test program by hand with the following command:

```bash
g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize this by allowing separate compilation of multiple files to create “object modules” (for C++, these files end in a “.o” extension) which can be linked together to form an executable in a separate stage of compilation. So, you could compile all of the files separately, using the “-c” flag, then link them together at the end:

```
g++ -g -c tester.cpp
g++ -g -c CHTNode.cpp
g++ -g -c ChainedHashTable.cpp
```

```
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it changes, both of the .cpp files should be recompiled. Unix provides the make utility to automate this process. The make utility takes a file called “Makefile” as input and automatically figures out which files need to be recompiled based on the rules that Makefile contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging directory. It consists of a sequence of “targets” followed by commands to be executed to create the targets. A line like:

```
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h \ 
CHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified since ChainedHashTable.o was created. (The “\” is required to indicate that the dependency list spans more than one line.) The following line in the Makefile is the command to execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if you have occasion to modify a Makefile or create one of your own, note that the first character of these command lines MUST be a tab, otherwise you’ll get very strange error messages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in the directory containing the files. When it finishes, type “ls” again. You should now see “.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type

```
ddd <executable-name>
```

where `<executable-name>` is replaced with the name of the executable you want to debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

**Setting a breakpoint**

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

**Examining variable values**

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

\[
t.table[0]\cdot t.size
\]

into the Argument window (next to the “( )” label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. Double-click on the key value. This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. Double click on the _M_dataplus field of key. _M_p in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

Stepping through a program

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variable. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert(“seventh”, 7)” to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump( )” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the formal parameters have been give for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

Debugging a “core dump”

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
ence. Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ run time system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type
   
   ./tester

   You should see some output from the program followed by the message

   Segmentation Fault (core dumped)

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type

   ddd tester core &

   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-
   Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name man-
   gling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “_ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHT-
   Node. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

**Exercise**

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
ddd Quick Reference

Running ddd
ddd <executable-name> where the name of your executable is substituted for <executable-name>

Viewing program output/typing console input
Choose View->Execution window

Setting a breakpoint
Click to the left of the line where you want to stop, then click the “Break” button.

Running the program
Click “Run” in the Command tool or choose Program->Run

Viewing variable values
“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

Viewing the contents of dynamically allocated arrays
Use “@” to specify the length of the array in the Argument area (e.g. a[0]@7)

Following pointers
Double-clicking on a pointer displays what it points to in the Data window.

Single stepping
“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.
“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

Executing to the end of a function/method
To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

Finding the source of a core dump
Type “ddd <executable> core” then choose Status->Backtrace.

Debugging using the ddd debugger
class CHTNode {
private:
    std::string key;
    int data;

public:
    CHTNode *next;

    CHTNode(std::string k, int payload); 
    std::string getKey(); 
    int getData();
};
```cpp
#include <string>
#include "CHTNode.h"
#include "KeyNotFoundException.h"
#include "DuplicateKeyException.h"

class ChainedHashTable {
public:
    ChainedHashTable(int slots);
    void insert (std::string k, int value);
    bool lookup (std::string k);
    int retrieve (std::string k);
    void remove (std::string k);
    void dump();

private:
    // Declaration for a dynamically allocated array of pointers to
    // CHTNodes -- this is the "array of linked lists" that serves
    // as our hash table.
    CHTNode **table;
    // size is the number of slots in table. C++ arrays don’t remember
    // this after they’re created, so we have to save it ourselves.
    int size;

    // Private helper method that generates the hash value for a
    // given key.
    int hash (std::string k);
};
```

// The "declaration" of the class, containing declaration of data members
// and prototypes for methods.
```
Debugging using the ddd debugger 11

// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include "ChainedHashTable.h"

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // “slots” elements.
    table = new CHTNode* [slots];
    size = slots;

    for (int i = 0; i < size; i++) {
        table[i] = 0;// Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    return found;
while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}

return found;

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey()) {
            found = true;
            break;
        }
        p = p->next;
    }
Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;
        prev->next = p->next;

        // This is necessary because C++ doesn’t automatically
        // garbage collect objects when there are no longer any
        // references to them.
        delete p;
    }
} else {
    throw KeyNotFoundException(k);
}

// This is essentially identical to program 14.1 in Sedgewick’s Algorithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
    int h = 0;
    int a = 127;

    for (int i = 0; i < k.length(); i++){
        h = (a * h + k[i]) % size;
    }

    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump(){
    std::cout << "Contents of table:" << std::endl << std::endl;

    for (int i = 0; i < size; i++) {
        std::cout << "Bucket " << i << ": " << std::endl;
        CHTNode *p = table[i];
    }
while (p != 0) {
    std::cout << "\tKey = " << p->getKey() << " value = " <<
              p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
}
// tester.cpp
#include <iostream>
#include <string>
#include "CHTNode.h"
#include "ChainedHashTable.h"

main() {
    // Declare a new hash table that stores ints as its data value.
    ChainedHashTable t(11);

    // Insert a number of values. We end up with some chains of
    // length 2.
    t.insert("first", 1);
    t.insert("second", 2);
    t.insert("frist", 32);
    t.insert("third", 3);
    t.insert("fourth", 4);
    t.insert("fifth", 5);
    t.insert("sixth", 6);
    t.insert("seventh", 7);

    // Test the response to inserting a duplicate value.
    try {
        t.insert("third", 9);
    } catch (DuplicateKeyException e) {
        // std::cerr is a separate output stream used for error reporting.
        // Even if the standard output stream is redirected to a file,
        // std::cerr's output appears on the screen.
        std::cerr << "Attempted to insert duplicate key " << e.key
                   << std::endl;
    }

    t.dump();

    int val = t.retrieve("sixth");
    std::cout << "retrieve(\"sixth\") returns " << val << std::endl;

    try {
        t.retrieve("bungle");
    } catch (KeyNotFoundException e) {
        std::cerr << "Didn't find key " << e.key << " in retrieve." << std::endl;
    }

    t.remove("sixth");

    std::cout << "After removing \"sixth\":" << std::endl;
try {
    t.remove("junk");
} catch (KeyNotFoundException e) {
    std::cerr << "Didn't find key " << e.key << " in remove." << std::endl << std::endl;
}

t.remove("fourth");
std::cout << "After removing ""fourth"":" << std::endl;
t.dump();

t.remove("seventh");
std::cout << "After removing ""seventh"":" << std::endl;
t.dump();
}
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window.
by clicking on the icon resembling a computer at the left end of the front panel.

Retrieving the files

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

cd cs301
mkdir debugging
cd debugging

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):

cp ~lbarnett/share/debugging/* .

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

CHTNode.cpp
CHTNode.h
ChainedHashTable.cpp
ChainedHashTable.h
DuplicateKeyException.h
KeyNotFoundException.h
Makefile
tester.cpp

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

Compiling the test program

You could compile the test program by hand with the following command:

`g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester`
If you were still working on the program and needed to recompile frequently, this would end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize this by allowing separate compilation of multiple files to create “object modules” (for C++, these files end in a “.o” extension) which can be linked together to form an executable in a separate stage of compilation. So, you could compile all of the files separately, using the “-c” flag, then link them together at the end:

```
g++ -g -c tester.cpp
g++ -g -c CHTNode.cpp
g++ -g -c ChainedHashTable.cpp
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it changes, both of the .cpp files should be recompiled. Unix provides the `make` utility to automate this process. The make utility takes a file called “Makefile” as input and automatically figures out which files need to be recompiled based on the rules that Makefile contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging directory. It consists of a sequence of “targets” followed by commands to be executed to create the targets. A line like:

```
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h \ 
CHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified since ChainedHashTable.o was created. (The “\” is required to indicate that the dependency list spans more than one line.) The following line in the Makefile is the command to execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if you have occasion to modify a Makefile or create one of your own, note that the first character of these command lines MUST be a tab, otherwise you’ll get very strange error messages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in the directory containing the files. When it finishes, type “ls” again. You should now see “.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type

   `ddd <executable-name>`

   where `<executable-name>` is replaced with the name of the executable you want to debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

Setting a breakpoint

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

Examining variable values

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

   `t.table[0]@t.size`

into the Argument window (next to the “( ):” label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. Double-click on the key value. This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. Double click on the _M_dataplus field of key. _M_p in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

**Stepping through a program**

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variable. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert(“seventh”, 7) to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump( )” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the the formal parameters have been give for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

**Debugging a “core dump”**

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
ence. Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ run time system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type
   ```
   ./tester
   ```
   You should see some output from the program followed by the message
   ```
   Segmentation Fault (core dumped)
   ```

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type
   ```
   ddd tester core &
   ```
   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name mangling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “_ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHTNode. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

**Exercise**

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
ddd Quick Reference

Running ddd

ddd <executable-name> where the name of your executable is substituted for <executable-name>

Viewing progam output/typing console input

Choose View->Execution window

Setting a breakpoint

Click to the left of the line where you want to stop, then click the “Break” button.

Running the program

Click “Run” in the Command tool or choose Program->Run

Viewing variable values

“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

Viewing the contents of dynamically allocated arrays

Use “@” to specify the length of the array in the Argument area (e.g. a[0]@7)

Following pointers

Double-clicking on a pointer displays what it points to in the Data window.

Single stepping

“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.

“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

Executing to the end of a function/method

To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

Finding the source of a core dump

Type “ddd <executable> core” then choose Status->Backtrace.
class CHTNode {
private:
    std::string key;
    int   data;

public:
    CHTNode *next;

    CHTNode(std::string k, int payload);
    std::string getKey();
    int getData();
};

#include "CHTNode.h"

// Constructor
CHTNode::CHTNode(std::string k, int payload) {
    key = k;
    data = payload;
}

// Return the key value stored in this node
std::string CHTNode::getKey() {
    return key;
}

// Return the data value stored in this node.
int CHTNode::getData() {
    return data;
}
A templated Chained Hash table. The keys are assumed to be strings. Duplicate keys are not allowed.

```cpp
#ifndef __ChainedHashTable__
#define __ChainedHashTable__

#include <string>
#include "CHTNode.h"
#include "KeyNotFoundException.h"
#include "DuplicateKeyException.h"

// The "declaration" of the class, containing declaration of data members
// and prototypes for methods.
class ChainedHashTable {
    public:
        ChainedHashTable(int slots);
        void insert (std::string k, int value);
        bool lookup (std::string k);
        int retrieve (std::string k);
        void remove(std::string k);
        void dump();

    private:
        // Declaration for a dynamically allocated array of pointers to
        // CHTNodes -- this is the "array of linked lists" that serves
        // as our hash table.
        CHTNode **table;
        // size is the number of slots in table. C++ arrays don’t remember
        // this after they’re created, so we have to save it ourselves.
        int size;

        // Private helper method that generates the hash value for a
        // given key.
        int hash (std::string k);

};

#endif
```
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include "ChainedHashTable.h"

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // “slots” elements.
    table = new CHTNode* [slots];
    size = slots;

    for (int i = 0; i < size; i++) {
        table[i] = 0;// Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];

    return found;
}
while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}

return found;

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey()) {
            found = true;
            break;
        }
        p = p->next;
    }
// Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;
        prev->next = p->next;
        // This is necessary because C++ doesn’t automatically
        // garbage collect objects when there are no longer any
        // references to them.
        delete p;
    }
} else {
    throw KeyNotFoundException(k);
}

// This is essentially identical to program 14.1 in Sedgewick’s Algo-
// rithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
    int h = 0;
    int a = 127;
    for (int i = 0; i < k.length(); i++) {
        h = (a * h + k[i]) % size;
    }
    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump() {
    std::cout << "Contents of table:" << std::endl << std::endl;
    for (int i = 0; i < size; i++) {
        std::cout << "Bucket " << i << ": " << std::endl;
        CHTNode *p = table[i];
while (p != 0) {
    std::cout << "\tKey = " << p->getKey() << " value = " << p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
main() {  
    // Declare a new hash table that stores ints as its data value.  
    ChainedHashTable t(11);  
  
    // Insert a number of values. We end up with some chains of  
    // length 2.  
    t.insert("first", 1);  
    t.insert("second", 2);  
    t.insert("frist", 32);  
    t.insert("third", 3);  
    t.insert("fourth", 4);  
    t.insert("fifth", 5);  
    t.insert("sixth", 6);  
    t.insert("seventh", 7);  
  
    // Test the response to inserting a duplicate value.  
    try {  
        t.insert("third", 9);  
    } catch (DuplicateKeyException e) {  
        // std::cerr is a separate output stream used for error reporting.  
        // Even if the standard output stream is redirected to a file,  
        // std::cerr's output appears on the screen.  
        std::cerr << "Attempted to insert duplicate key " << e.key  
                   << std::endl;  
    }  
  
    t.dump();  
  
    int val = t.retrieve("sixth");  
    std::cout << "retrieve("sixth") returns " << val << std::endl;  
    try {  
        t.retrieve("bungle");  
    } catch (KeyNotFoundException e) {  
        std::cerr << "Didn't find key " << e.key << " in retrieve." << std::endl;  
    }  
  
    t.remove("sixth");  
  
    std::cout << "After removing ""sixth":" << std::endl;
t.dump();

try {
    t.remove("junk");
} catch (KeyNotFoundException e) {
    std::cerr << "Didn't find key " << e.key << " in remove." <<
                std::endl << std::endl;
}

t.remove("fourth");
std::cout << "After removing \"fourth\":" << std::endl;
t.dump();

t.remove("seventh");
std::cout << "After removing \"seventh\":" << std::endl;
t.dump();
}
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```
cd cs301
mkdir debugging
cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):

```
cp ~lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

```
CHTNode.cpp
CHTNode.h
ChainedHashTable.cpp
ChainedHashTable.h
DuplicateKeyException.h
KeyNotFoundException.h
Makefile
tester.cpp
```

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You could compile the test program by hand with the following command:

```
g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize this by allowing separate compilation of multiple files to create “object modules” (for C++, these files end in a “.o” extension) which can be linked together to form an executable in a separate stage of compilation. So, you could compile all of the files separately, using the “-c” flag, then link them together at the end:

```bash
g++ -g -c tester.cpp
g++ -g -c CHTNode.cpp
g++ -g -c ChainedHashTable.cpp
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it changes, both of the .cpp files should be recompiled. Unix provides the make utility to automate this process. The make utility takes a file called “Makefile” as input and automatically figures out which files need to be recompiled based on the rules that Makefile contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging directory. It consists of a sequence of “targets” followed by commands to be executed to create the targets. A line like:

```
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h \ 
CHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified since ChainedHashTable.o was created. (The “\” is required to indicate that the dependency list spans more than one line.) The following line in the Makefile is the command to execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if you have occasion to modify a Makefile or create one of your own, note that the first character of these command lines MUST be a tab, otherwise you’ll get very strange error messages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in the directory containing the files. When it finishes, type “ls” again. You should now see “.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type

   ```
ddd <executable-name>
   
   where <executable-name> is replaced with the name of the executable you want to debug. For this exercise, type
   ```
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

### Setting a breakpoint

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

### Examining variable values

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

   t.table[0]@t.size

into the Argument window (next to the “( )” label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. Double-click on the key value. This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. Double click on the _M_dataplus field of key. _M_p in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

**Stepping through a program**

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variable. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert(“seventh”, 7) to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump( )” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the the formal parameters have been give for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

**Debugging a “core dump”**

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
ence. Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ run time system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type

   ```
   ./tester
   ```

   You should see some output from the program followed by the message

   ```
   Segmentation Fault (core dumped)
   ```

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type

   ```
   ddd tester core &
   ```

   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name mangling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “_ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHTNode. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

**Exercise**

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
**ddd Quick Reference**

**Running ddd**

ddd `<executable-name>` where the name of your executable is substituted for `<executable-name>`

**Viewing program output/typing console input**

Choose View->Execution window

**Setting a breakpoint**

Click to the left of the line where you want to stop, then click the “Break” button.

**Running the program**

Click “Run” in the Command tool or choose Program->Run

**Viewing variable values**

“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

**Viewing the contents of dynamically allocated arrays**

Use “@” to specify the length of the array in the Argument area (e.g. a[0]@7)

**Following pointers**

Double-clicking on a pointer displays what it points to in the Data window.

**Single stepping**

“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.

“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

**Executing to the end of a function/method**

To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

**Finding the source of a core dump**

Type “ddd `<executable> core” then choose Status->Backtrace.
// File: CHTNode.h
// Author: Lewis Barnett

// A Node to be used in a Chained Hash Table. It contains a key, which
// is a string, and a data payload, which is an int.
#ifndef __CHTNode__
#define __CHTNode__

#include <string>

class CHTNode {
private:
    std::string key;
    int   data;

public:
    CHTNode *next;

    CHTNode(std::string k, int payload);
    std::string getKey();
    int getData();
};
#endif

// Constructor
CHTNode::CHTNode(std::string k, int payload) {
    key = k;
    data = payload;
}

// Return the key value stored in this node
std::string CHTNode::getKey() {
    return key;
}

// Return the data value stored in this node.
int CHTNode::getData() {
    return data;
}
A templated Chained Hash table. The keys are assumed to be strings. // Duplicate keys are not allowed.

#ifndef __ChainedHashTable__
define __ChainedHashTable__

#include <string>
#include "CHTNode.h"
#include "KeyNotFoundException.h"
#include "DuplicateKeyException.h"

// The “declaration” of the class, containing declaration of data members
// and prototypes for methods.
class ChainedHashTable {
public:
   ChainedHashTable(int slots);
   void insert (std::string k, int value);
   bool lookup (std::string k);
   int retrieve (std::string k);
   void remove(std::string k);
   void dump();

private:
   // Declaration for a dynamically allocated array of pointers to
   // CHTNodes -- this is the “array of linked lists” that serves
   // as our hash table.
   CHTNode **table;
   // size is the number of slots in table. C++ arrays don’t remember
   // this after they’re created, so we have to save it ourselves.
   int size;

   // Private helper method that generates the hash value for a
   // given key.
   int hash (std::string k);
};

#endif
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The "implementation" of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include "ChainedHashTable.h"

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // "slots" elements.
    table = new CHTNode* [slots];
    size = slots;

    for (int i = 0; i < size; i++) {
        table[i] = 0;// Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];

while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}

return found;

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey()) {
            found = true;
            break;
        }
        p = p->next;
    }

// Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;
        prev->next = p->next;

        // This is necessary because C++ doesn’t automatically
        // garbage collect objects when there are no longer any
        // references to them.
        delete p;
    }
} else {
    throw KeyNotFoundException(k);
}

// This is essentially identical to program 14.1 in Sedgewick’s Algo-
// rithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
    int h = 0;
    int a = 127;

    for (int i = 0; i < k.length(); i++){
        h = (a * h + k[i]) % size;
    }

    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump(){
    std::cout << “Contents of table:” << std::endl << std::endl;

    for (int i = 0; i < size; i++) {
        std::cout << “Bucket “ << i << “: “ << std::endl;
        CHTNode *p = table[i];
    }
while (p != 0) {
    std::cout << "\t Key = " << p->getKey() << " value = " << p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
// tester.cpp
#include <iostream>
#include <string>
#include "CHTNode.h"
#include "ChainedHashTable.h"

main() {
    // Declare a new hash table that stores ints as its data value.
    ChainedHashTable t(11);

    // Insert a number of values. We end up with some chains of
    // length 2.
    t.insert("first", 1);
    t.insert("second", 2);
    t.insert("frist", 32);
    t.insert("third", 3);
    t.insert("fourth", 4);
    t.insert("fifth", 5);
    t.insert("sixth", 6);
    t.insert("seventh", 7);

    // Test the response to inserting a duplicate value.
    try {
        t.insert("third", 9);
    } catch (DuplicateKeyException e) {
        // std::cerr is a separate output stream used for error reporting.
        // Even if the standard output stream is redirected to a file,
        // std::cerr’s output appears on the screen.
        std::cerr << "Attempted to insert duplicate key " << e.key
                   << std::endl;
    }

t.dump();

    int val = t.retrieve("sixth");
    std::cout << "retrieve(\"sixth\") returns " << val << std::endl;

    try {
        t.retrieve("bungle");
    } catch (KeyNotFoundException e) {
        std::cerr << "Didn’t find key " << e.key << " in retrieve." << std::endl << std::endl;
    }

t.remove("sixth");

    std::cout << "After removing \"sixth\":" << std::endl;
t.dump();

try {
    t.remove("junk");
} catch (KeyNotFoundException e) {
    std::cerr << "Didn’t find key " << e.key << " in remove." << 
              std::endl << std::endl;
}

t.remove("fourth");
std::cout << "After removing \"fourth\":" << std::endl;
t.dump();

t.remove("seventh");
std::cout << "After removing \"seventh\":" << std::endl;
t.dump();
}
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```
  cd cs301
  mkdir debugging
  cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):

```
  cp ~lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

- CHTNode.cpp
- CHTNode.h
- ChainedHashTable.cpp
- ChainedHashTable.h
- DuplicateKeyException.h
- KeyNotFoundException.h
- Makefile
- tester.cpp

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You *could* compile the test program by hand with the following command:

```
  g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize this by allowing separate compilation of multiple files to create “object modules” (for C++, these files end in a “.o” extension) which can be linked together to form an executable in a separate stage of compilation. So, you could compile all of the files separately, using the “-c” flag, then link them together at the end:

```
g++ -g -c tester.cpp
g++ -g -c CHTNode.cpp
g++ -g -c ChainedHashTable.cpp
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it changes, both of the .cpp files should be recompiled. Unix provides the make utility to automate this process. The make utility takes a file called “Makefile” as input and automatically figures out which files need to be recompiled based on the rules that Makefile contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging directory. It consists of a sequence of “targets” followed by commands to be executed to create the targets. A line like:

```
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h ChHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified since ChainedHashTable.o was created. (The “\” is required to indicate that the dependency list spans more than one line.) The following line in the Makefile is the command to execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if you have occasion to modify a Makefile or create one of your own, note that the first character of these command lines MUST be a tab, otherwise you’ll get very strange error messages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in the directory containing the files. When it finishes, type “ls” again. You should now see “.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type
   ```
   ddd <executable-name>
   ```
   where `<executable-name>` is replaced with the name of the executable you want to debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

**Setting a breakpoint**

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

**Examining variable values**

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

   t.table[0]@t.size

into the Argument window (next to the “( ):” label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. Double-click on the key value. This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. Double click on the _M_dataplus field of key. _M_p in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

Stepping through a program

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variable. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert("seventh", 7) to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump()” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the the formal parameters have been give for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

Debugging a “core dump”

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
ence. Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ run time system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type

   ./tester

   You should see some output from the program followed by the message

   Segmentation Fault (core dumped)

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type

   ddd tester core &

   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name mangling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “_ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHT-Node. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

Exercise

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
**ddd Quick Reference**

**Running ddd**

`ddd <executable-name>` where the name of your executable is substituted for `<executable-name>`

**Viewing program output/typing console input**

Choose View->Execution window

**Setting a breakpoint**

Click to the left of the line where you want to stop, then click the “Break” button.

**Running the program**

Click “Run” in the Command tool or choose Program->Run

**Viewing variable values**

“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

**Viewing the contents of dynamically allocated arrays**

Use “@” to specify the length of the array in the Argument area (e.g. a[0]@7)

**Following pointers**

Double-clicking on a pointer displays what it points to in the Data window.

**Single stepping**

“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.

“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

**Executing to the end of a function/method**

To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

**Finding the source of a core dump**

Type “ddd <executable> core” then choose Status->Backtrace.
#ifndef __CHTNode__
#define __CHTNode__

#include <string>

class CHTNode {
private:
    std::string key;
    int data;

public:
    CHTNode *next;

    CHTNode(std::string k, int payload);
    std::string getKey();
    int getData();
};
#endif

#include "CHTNode.h"

// Constructor
CHTNode::CHTNode(std::string k, int payload) {
    key = k;
    data = payload;
}

// Return the key value stored in this node
std::string CHTNode::getKey() {
    return key;
}

// Return the data value stored in this node.
int CHTNode::getData() {
    return data;
}
A templated Chained Hash table. The keys are assumed to be strings. Duplicate keys are not allowed.

#ifndef __ChainedHashTable__
define __ChainedHashTable__

#include <string>
#include "CHTNode.h"
#include "KeyNameNotFoundException.h"
#include "DuplicateKeyException.h"

// The “declaration” of the class, containing declaration of data members and prototypes for methods.
class ChainedHashTable {
public:
    ChainedHashTable(int slots);
    void insert (std::string k, int value);
    bool lookup (std::string k);
    int retrieve (std::string k);
    void remove(std::string k);
    void dump();

private:
    // Declaration for a dynamically allocated array of pointers to CHTNodes -- this is the “array of linked lists” that serves as our hash table.
    CHTNode **table;
    // size is the number of slots in table. C++ arrays don’t remember this after they’re created, so we have to save it ourselves.
    int size;

    // Private helper method that generates the hash value for a given key.
    int hash (std::string k);
};

#endif
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include "ChainedHashTable.h"

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // “slots” elements.
    table = new CHTNode* [slots];
    size = slots;
    for (int i = 0; i < size; i++) {
        table[i] = 0;// Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (p) {
        if (p->key == k) {
            found = true;
            break;
        }
        p = p->next;
    }
    return found;
}
while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}
return found;

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey()) {
            found = true;
            break;
        }
        p = p->next;
    }
// Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;

        prev->next = p->next;

        // This is necessary because C++ doesn’t automatically
        // garbage collect objects when there are no longer any
        // references to them.
        delete p;
    }
} else {
    throw KeyNotFoundException(k);
}

// This is essentially identical to program 14.1 in Sedgewick’s Algo-
// rithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
    int h = 0;
    int a = 127;

    for (int i = 0; i < k.length(); i++){
        h = (a * h + k[i]) % size;
    }

    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump(){
    std::cout << "Contents of table:" << std::endl;

    for (int i = 0; i < size; i++) {
        std::cout << "Bucket " << i << " : " << std::endl;
        CHTNode *p = table[i];
while (p != 0) {
    std::cout << "Key = " << p->getKey() << " value = " << p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
}
// tester.cpp
#include <iostream>
#include <string>
#include "CHTNode.h"
#include "ChainedHashTable.h"

main() {
   // Declare a new hash table that stores ints as its data value.
   ChainedHashTable t(11);

   // Insert a number of values. We end up with some chains of
   // length 2.
   t.insert("first", 1);
   t.insert("second", 2);
   t.insert("frist", 32);
   t.insert("third", 3);
   t.insert("fourth", 4);
   t.insert("fifth", 5);
   t.insert("sixth", 6);
   t.insert("seventh", 7);

   // Test the response to inserting a duplicate value.
   try {
      t.insert("third", 9);
   } catch (DuplicateKeyException e) {
      // std::cerr is a separate output stream used for error reporting.
      // Even if the standard output stream is redirected to a file,
      // std::cerr's output appears on the screen.
      std::cerr << "Attempted to insert duplicate key " << e.key
               << std::endl;
   }

   t.dump();

   int val = t.retrieve("sixth");
   std::cout << "retrieve("sixth") returns " << val << std::endl;

   try {
      t.retrieve("bungle");
   } catch (KeyNotFoundException e) {
      std::cerr << "Didn't find key " << e.key << " in retrieve." << std::endl;
   }

   t.remove("sixth");

   std::cout << "After removing "sixth"): " << std::endl;
t.dump();

try {
    t.remove("junk");
} catch (KeyNotFoundException e) {
    std::cerr << "Didn’t find key " << e.key << " in remove." <<
        std::endl << std::endl;
}

t.remove("fourth");
std::cout << "After removing ""fourth":" << std::endl;
t.dump();

(t.remove("seventh");
std::cout << "After removing ""seventh":" << std::endl;
t.dump();
)
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```
cd cs301
mkdir debugging
cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):

```
cp ~lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

```
CHTNode.cpp
CHTNode.h
ChainedHashTable.cpp
ChainedHashTable.h
DuplicateKeyException.h
KeyNotFoundException.h
Makefile
tester.cpp
```

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You *could* compile the test program by hand with the following command:

```
g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would
end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean
that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize
this by allowing separate compilation of multiple files to create “object modules” (for
C++, these files end in a “.o” extension) which can be linked together to form an execut-
able in a separate stage of compilation. So, you could compile all of the files separately,
using the “-c” flag, then link them together at the end:

```bash
g++ -g -c tester.cpp
g++ -g -c CHTNode.cpp
g++ -g -c ChainedHashTable.cpp
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the
final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For
example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it
changes, both of the .cpp files should be recompiled. Unix provides the make utility to
automate this process. The make utility takes a file called “Makefile” as input and auto-
matically figures out which files need to be recompiled based on the rules that Makefile
contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging direc-
tory. It consists of a sequence of “targets” followed by commands to be executed to create
the targets. A line like:

```make
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h \
CHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three
files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified
since ChainedHashTable.o was created. (The “\" is required to indicate that the depen-
dency list spans more than one line.) The following line in the Makefile is the command to
execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if
you have occasion to modify a Makefile or create one of your own, note that the first char-
acter of these command lines MUST be a tab, otherwise you’ll get very strange error mes-
sages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in
the directory containing the files. When it finishes, type “ls” again. You should now see
“.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable
we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type
   ```bash
   ddd <executable-name>
   ```
   where `<executable-name>` is replaced with the name of the executable you want to
debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

Setting a breakpoint

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

Examining variable values

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

\[
t.table[0]@t.size
\]

into the Argument window (next to the “( );” label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. Double-click on the key value. This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. Double click on the _M_dataplus field of key. _M_p in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

Stepping through a program

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variables. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert("seventh", 7) to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump( )” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the the formal parameters have been given for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

Debugging a “core dump”

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
ence. Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ run time system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type
   ./tester
   You should see some output from the program followed by the message
   Segmentation Fault (core dumped)

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type
   ddd tester core &

   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name mangling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “_ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHTNode. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

Exercise

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
**ddd Quick Reference**

**Running ddd**
ddd `<executable-name>` where the name of your executable is substituted for `<executable-name>`

** Viewing program output/typing console input**
Choose View->Execution window

**Setting a breakpoint**
Click to the left of the line where you want to stop, then click the “Break” button.

**Running the program**
Click “Run” in the Command tool or choose Program->Run

**Viewing variable values**
“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

**Viewing the contents of dynamically allocated arrays**
Use “@” to specify the length of the array in the Argument area (e.g. a[0]@7)

**Following pointers**
Double-clicking on a pointer displays what it points to in the Data window.

**Single stepping**
“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.
“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

**Executing to the end of a function/method**
To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

**Finding the source of a core dump**
Type “ddd `<executable>` core” then choose Status->Backtrace.
// File: CHTNode.h
// Author: Lewis Barnett

// A Node to be used in a Chained Hash Table. It contains a key, which
// is a string, and a data payload, which is an int.
#ifndef __CHTNode__
#define __CHTNode__

#include <string>

class CHTNode {
    private:
        std::string key;
        int data;

    public:
        CHTNode *next;

        CHTNode(std::string k, int payload);
        std::string getKey();
        int getData();
    }
#endif

#include "CHTNode.h"

// Constructor
CHTNode::CHTNode(std::string k, int payload) {
    key = k;
    data = payload;
}

// Return the key value stored in this node
std::string CHTNode::getKey() {
    return key;
}

// Return the data value stored in this node.
in CHTNode::getData() {
    return data;
}
// File: ChainedHashTable.h
// Author: Lewis Barnett

// A templated Chained Hash table. The keys are assumed to be strings.
// Duplicate keys are not allowed.

#ifndef __ChainedHashTable__
#define __ChainedHashTable__

#include <string>
#include "CHTNode.h"
#include "KeyNotFoundException.h"
#include "DuplicateKeyException.h"

// The "declaration" of the class, containing declaration of data members
// and prototypes for methods.
class ChainedHashTable {
 public:
  ChainedHashTable(int slots);
  void insert (std::string k, int value);
  bool lookup (std::string k);
  int retrieve (std::string k);
  void remove(std::string k);
  void dump();

 private:
  // Declaration for a dynamically allocated array of pointers to
  // CHTNodes -- this is the "array of linked lists" that serves
  // as our hash table.
  CHTNode **table;
  // size is the number of slots in table. C++ arrays don’t remember
  // this after they’re created, so we have to save it ourselves.
  int size;

  // Private helper method that generates the hash value for a
  // given key.
  int hash (std::string k);

};

#endif
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include “ChainedHashTable.h”

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // “slots” elements.
    table = new CHTNode* [slots];
    size = slots;

    for (int i = 0; i < size; i++) {
        table[i] = 0; // Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup(std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];

    while (p && !found) {
        if (p->key == k) {
            found = true;
        }
        p = p->next;
    }

    return found;
}
while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}

return found;
}

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey()) {
            found = true;
            break;
        }
        p = p->next;
    }
// Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;

        prev->next = p->next;

        // This is necessary because C++ doesn’t automatically
        // garbage collect objects when there are no longer any
        // references to them.
        delete p;
    }
} else {
    throw KeyNotFoundException(k);
}
}

// This is essentially identical to program 14.1 in Sedgewick’s Algo-
// rithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
    int h = 0;
    int a = 127;

    for (int i = 0; i < k.length(); i++)
        h = (a * h + k[i]) % size;

    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump(){
    std::cout << "Contents of table:" << std::endl << std::endl;

    for (int i = 0; i < size; i++) {
        std::cout << "Bucket " << i << " : " << std::endl;
        CHTNode *p = table[i];
while (p != 0) {
    std::cout << "\tKey = " << p->getKey() << " value = " << p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
// tester.cpp
#include <iostream>
#include <string>
#include "CHTNode.h"
#include "ChainedHashTable.h"

main() {
    // Declare a new hash table that stores ints as its data value.
    ChainedHashTable t(11);

    // Insert a number of values. We end up with some chains of
    // length 2.
    t.insert("first", 1);
    t.insert("second", 2);
    t.insert("first", 3);
    t.insert("third", 3);
    t.insert("fourth", 4);
    t.insert("fifth", 5);
    t.insert("sixth", 6);
    t.insert("seventh", 7);

    // Test the response to inserting a duplicate value.
    try {
        t.insert("third", 9);
    } catch (DuplicateKeyException e) {
        // std::cerr is a separate output stream used for error reporting.
        // Even if the standard output stream is redirected to a file,
        // std::cerr's output appears on the screen.
        std::cerr << "Attempted to insert duplicate key " << e.key
                   << std::endl;
    }

    t.dump();

    int val = t.retrieve("sixth");
    std::cout << "retrieve("sixth") returns " << val << std::endl;

    try {
        t.retrieve("bungle");
    } catch (KeyNotFoundException e) {
        std::cerr << "Didn't find key " << e.key << " in retrieve." << std::endl;
    }

    t.remove("sixth");

    std::cout << "After removing "sixth":" << std::endl;
}
t.dump();

try {
    t.remove("junk");
} catch (KeyNotFoundException e) {
    std::cerr << "Didn’t find key " << e.key << " in remove."
               << std::endl << std::endl;
}

t.remove("fourth");
std::cout << "After removing \"fourth\":" << std::endl;
  t.dump();

    t.remove("seventh");
std::cout << "After removing \"seventh\":" << std::endl;
  t.dump();
}
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window

---

Debugging using the ddd debugger
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```bash
cd cs301
mkdir debugging
cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):

```bash
cp ~lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

- CHTNode.cpp
- CHTNode.h
- ChainedHashTable.cpp
- ChainedHashTable.h
- DuplicateKeyException.h
- KeyNotFoundException.h
- Makefile
- tester.cpp

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You *could* compile the test program by hand with the following command:

```bash
g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize this by allowing separate compilation of multiple files to create “object modules” (for C++, these files end in a “.o” extension) which can be linked together to form an executable in a separate stage of compilation. So, you could compile all of the files separately, using the “-c” flag, then link them together at the end:

```
g++ -g -c tester.cpp
```
```
g++ -g -c CHTNode.cpp
```
```
g++ -g -c ChainedHashTable.cpp
```
```
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it changes, both of the .cpp files should be recompiled. Unix provides the make utility to automate this process. The make utility takes a file called “Makefile” as input and automatically figures out which files need to be recompiled based on the rules that Makefile contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging directory. It consists of a sequence of “targets” followed by commands to be executed to create the targets. A line like:

```
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h CHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified since ChainedHashTable.o was created. (The “\” is required to indicate that the dependency list spans more than one line.) The following line in the Makefile is the command to execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if you have occasion to modify a Makefile or create one of your own, note that the first character of these command lines MUST be a tab, otherwise you’ll get very strange error messages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in the directory containing the files. When it finishes, type “ls” again. You should now see “.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type

   `ddd <executable-name>`

   where `<executable-name>` is replaced with the name of the executable you want to debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

### Setting a breakpoint

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.
2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.
3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

### Examining variable values

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.
2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

   t.table[0]@t.size

into the Argument window (next to the “( )” label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

   At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. Double-click on the key value. This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. Double click on the _M_dataplus field of key. _M_p in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.”) I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

Stepping through a program

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variable. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert(“seventh”, 7)” to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump( )” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the the formal parameters have been give for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

Debugging a “core dump”

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
ence. Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ runtime system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type
   ```
   ./tester
   ```
   You should see some output from the program followed by the message
   ```
   Segmentation Fault (core dumped)
   ```

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type
   ```
   ddd tester core &
   ```
   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name mangling” which adds additional information to the names of methods based on what data type you’ve used for the template. For example, what I actually saw for the method name was “_ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHTNode. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

**Exercise**

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
**ddd Quick Reference**

**Running ddd**

ddd `<executable-name>` where the name of your executable is substituted for `<executable-name>`

**Viewing program output/typing console input**

Choose View->Execution window

**Setting a breakpoint**

Click to the left of the line where you want to stop, then click the “Break” button.

**Running the program**

Click “Run” in the Command tool or choose Program->Run

**Viewing variable values**

“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

**Viewing the contents of dynamically allocated arrays**

Use “@” to specify the length of the array in the Argument area (e.g. `a[0]@7`)

**Following pointers**

Double-clicking on a pointer displays what it points to in the Data window.

**Single stepping**

“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.

“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

**Executing to the end of a function/method**

To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

**Finding the source of a core dump**

Type “ddd `<executable> core” then choose Status->Backtrace.”
#include <string>

class CHTNode {
private:
    std::string key;
    int data;

public:
    CHTNode *next;

    CHTNode(std::string k, int payload);
    std::string getKey();
    int getData();
};

CHTNode::CHTNode(std::string k, int payload) {
    key = k;
    data = payload;
}

std::string CHTNode::getKey() {
    return key;
}

int CHTNode::getData() {
    return data;
}
// File: ChainedHashTable.h
// Author: Lewis Barnett

// A templated Chained Hash table. The keys are assumed to be strings.
// Duplicate keys are not allowed.

#ifndef __ChainedHashTable__
#define __ChainedHashTable__

#include <string>
#include "CHTNode.h"
#include "KeyNotFoundException.h"
#include "DuplicateKeyException.h"

// The "declaration" of the class, containing declaration of data members
// and prototypes for methods.
class ChainedHashTable {

public:
    ChainedHashTable(int slots);
    void insert (std::string k, int value);
    bool lookup (std::string k);
    int retrieve (std::string k);
    void remove(std::string k);
    void dump();

private:
    // Declaration for a dynamically allocated array of pointers to
    // CHTNodes -- this is the "array of linked lists" that serves
    // as our hash table.
    CHTNode **table;
    // size is the number of slots in table. C++ arrays don’t remember
    // this after they’re created, so we have to save it ourselves.
    int size;

    // Private helper method that generates the hash value for a
    // given key.
    int hash (std::string k);

};

#endif
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include “ChainedHashTable.h”

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // “slots” elements.
    table = new CHTNode* [slots];
    size = slots;

    for (int i = 0; i < size; i++) {
        table[i] = 0;// Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];

    return false;
}
while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}

return found;
}

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey()) {
            found = true;
            break;
        }
        p = p->next;
    }
Debugging using the ddd debugger

Now, we need a pointer to the preceding node in the list

if (found) {
  // Deleting the first key in the bucket is a special case
  if (p == table[bucket]) {
    table[bucket] = p->next;
  } else {
    // We scan the list a second time here; a more efficient
    // implementation would keep track of a trailing pointer
    // in the initial loop.
    CHTNode *prev = table[bucket];
    while (prev->next != 0 && prev->next != p)
      prev = prev->next;

    prev->next = p->next;

    // This is necessary because C++ doesn’t automatically
    // garbage collect objects when there are no longer any
    // references to them.
    delete p;
  }
} else {
  throw KeyNotFoundException(k);
}

// This is essentially identical to program 14.1 in Sedgewick’s Algorithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
  int h = 0;
  int a = 127;

  for (int i = 0; i < k.length(); i++){
    h = (a * h + k[i]) % size;
  }

  return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump(){
  std::cout << “Contents of table:” << std::endl;

  for (int i = 0; i < size; i++) {
    std::cout << “Bucket “ << i << “: “ << std::endl;
    CHTNode *p = table[i];
  }
}
while (p != 0) {
    std::cout << "\tKey = " << p->getKey() << " value = " <<
    p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
}
// tester.cpp
#include <iostream>
#include <string>
#include “CHTNode.h”
#include “ChainedHashTable.h”

main() {
    // Declare a new hash table that stores ints as its data value.
    ChainedHashTable t(11);

    // Insert a number of values. We end up with some chains of
    // length 2.
    t.insert(“first”, 1);
    t.insert(“second”, 2);
    t.insert(“frist”, 32);
    t.insert(“third”, 3);
    t.insert(“fourth”, 4);
    t.insert(“fifth”, 5);
    t.insert(“sixth”, 6);
    t.insert(“seventh”, 7);

    // Test the response to inserting a duplicate value.
    try {
        t.insert(“third”, 9);
    } catch (DuplicateKeyException e) {
        // std::cerr is a separate output stream used for error reporting.
        // Even if the standard output stream is redirected to a file,
        // std::cerr’s output appears on the screen.
        std::cerr << “Attempted to insert duplicate key “ << e.key
                  << std::endl;
    }

    t.dump();

    int val = t.retrieve(“sixth”);
    std::cout << “retrieve("sixth") returns “ << val << std::endl;

    try {
        t.retrieve(“bungle”);
    } catch (KeyNotFoundException e) {
        std::cerr << “Didn’t find key “ << e.key << “ in retrieve.” << std::endl << std::endl;
    }

    t.remove(“sixth”);

    std::cout << “After removing “ << std::endl;
}
t.dump();

try {
    t.remove("junk");
} catch (KeyNotFoundException e) {
    std::cerr << "Didn’t find key " << e.key << " in remove."
    std::endl << std::endl;
}

t.remove("fourth");
std::cout << "After removing "fourth":" << std::endl;
t.dump();

t.remove("seventh");
std::cout << "After removing "seventh":" << std::endl;
t.dump();
}
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window.
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```
cd cs301
mkdir debugging
cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):

```
cp ~lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

```
CHTNode.cpp
CHTNode.h
ChainedHashTable.cpp
ChainedHashTable.h
DuplicateKeyException.h
KeyNotFoundException.h
Makefile
tester.cpp
```

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You *could* compile the test program by hand with the following command:

```
g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize this by allowing separate compilation of multiple files to create “object modules” (for C++, these files end in a “.o” extension) which can be linked together to form an executable in a separate stage of compilation. So, you could compile all of the files separately, using the “-c” flag, then link them together at the end:

```
g++ -g -c tester.cpp
g++ -g -c CHTNode.cpp
g++ -g -c ChainedHashTable.cpp
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it changes, both of the .cpp files should be recompiled. Unix provides the `make` utility to automate this process. The make utility takes a file called “Makefile” as input and automatically figures out which files need to be recompiled based on the rules that Makefile contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging directory. It consists of a sequence of “targets” followed by commands to be executed to create the targets. A line like:

```
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h \ 
   CHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified since ChainedHashTable.o was created. (The “\” is required to indicate that the dependency list spans more than one line.) The following line in the Makefile is the command to execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if you have occasion to modify a Makefile or create one of your own, note that the first character of these command lines MUST be a tab, otherwise you’ll get very strange error messages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in the directory containing the files. When it finishes, type “ls” again. You should now see “.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type

```
  ddd <executable-name>
```

where `<executable-name>` is replaced with the name of the executable you want to debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

**Setting a breakpoint**

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

**Examining variable values**

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

   t.table[0]@t.size

into the Argument window (next to the “( );” label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. Double-click on the key value. This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. Double click on the _M_dataplus field of key. _M_p in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

**Stepping through a program**

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variable. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert(“seventh”, 7)” to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump()” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the the formal parameters have been give for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

**Debugging a “core dump”**

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
ence. Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ run time system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type

   ./tester

   You should see some output from the program followed by the message

   Segmentation Fault (core dumped)

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type

   ddd tester core &

   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name mangling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHTNode. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

Exercise

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
**ddd Quick Reference**

**Running ddd**

ddd <executable-name> where the name of your executable is substituted for <executable-name>

**Viewing program output/typing console input**

Choose View->Execution window

**Setting a breakpoint**

Click to the left of the line where you want to stop, then click the “Break” button.

**Running the program**

Click “Run” in the Command tool or choose Program->Run

**Viewing variable values**

“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

**Viewing the contents of dynamically allocated arrays**

Use “@” to specify the length of the array in the Argument area (e.g. a[0]@7)

**Following pointers**

Double-clicking on a pointer displays what it points to in the Data window.

**Single stepping**

“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.

“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

**Executing to the end of a function/method**

To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

**Finding the source of a core dump**

Type “ddd <executable> core” then choose Status->Backtrace.
class CHTNode {
private:
    std::string key;
    int   data;

public:
    CHTNode *next;

    CHTNode(std::string k, int payload);
    std::string getKey();
    int getData();
};

#include <string>

CHTNode::CHTNode(std::string k, int payload) {
    key = k;
    data = payload;
}

std::string CHTNode::getKey() { return key; }

int CHTNode::getData() { return data; }
// File: ChainedHashTable.h
// Author: Lewis Barnett

// A templated Chained Hash table. The keys are assumed to be strings.
// Duplicate keys are not allowed.

#ifndef __ChainedHashTable__
#define __ChainedHashTable__

#include <string>
#include "CHTNode.h"
#include "KeyNotFoundException.h"
#include "DuplicateKeyException.h"

// The “declaration” of the class, containing declaration of data members
// and prototypes for methods.
class ChainedHashTable {
    public:
        ChainedHashTable(int slots);
        void insert (std::string k, int value);
        bool lookup (std::string k);
        int retrieve (std::string k);
        void remove(std::string k);
        void dump();

    private:
        // Declaration for a dynamically allocated array of pointers to
        // CHTNodes -- this is the “array of linked lists” that serves
        // as our hash table.
        CHTNode **table;
        // size is the number of slots in table. C++ arrays don’t remember
        // this after they’re created, so we have to save it ourselves.
        int    size;

        // Private helper method that generates the hash value for a
        // given key.
        int    hash (std::string k);

};

#endif
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include “ChainedHashTable.h”

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // “slots” elements.
    table = new CHTNode* [slots];
    size = slots;

    for (int i = 0; i < size; i++) {
        table[i] = 0; // Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    ”}
while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}

return found;

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey()) {
            found = true;
            break;
        }
        p = p->next;
    }
// Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;
        prev->next = p->next;
    }
}
else {
    throw KeyNotFoundException(k);
}

// This is essentially identical to program 14.1 in Sedgewick’s Algorithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k) {
    int h = 0;
    int a = 127;
    for (int i = 0; i < k.length(); i++) {
        h = (a * h + k[i]) % size;
    }
    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump() {
    std::cout << "Contents of table:" << std::endl << std::endl;
    for (int i = 0; i < size; i++) {
        std::cout << "Bucket " << i << " : " << std::endl;
        CHTNode *p = table[i];
    }
}
while (p != 0) {
    std::cout << "\tKey = " << p->getKey() << "  value = " <<
              p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
}
// tester.cpp
#include <iostream>
#include <string>
#include "CHTNode.h"
#include "ChainedHashTable.h"

main() {
    // Declare a new hash table that stores ints as its data value.
    ChainedHashTable t(11);

    // Insert a number of values. We end up with some chains of
    // length 2.
    t.insert("first", 1);
    t.insert("second", 2);
    t.insert("frist", 32);
    t.insert("third", 3);
    t.insert("fourth", 4);
    t.insert("fifth", 5);
    t.insert("sixth", 6);
    t.insert("seventh", 7);

    // Test the response to inserting a duplicate value.
    try {
        t.insert("third", 9);
    } catch (DuplicateKeyException e) {
        // std::cerr is a separate output stream used for error reporting.
        // Even if the standard output stream is redirected to a file,
        // std::cerr's output appears on the screen.
        std::cerr << "Attempted to insert duplicate key " << e.key
                  << std::endl;
    }
    t.dump();

    int val = t.retrieve("sixth");
    std::cout << "retrieve("sixth") returns " << val << std::endl;

    try {
        t.retrieve("bungle");
    } catch (KeyNotFoundException e) {
        std::cerr << "Didn't find key " << e.key << " in retrieve." << std::endl;
    }
    t.remove("sixth");

    std::cout << "After removing "sixth":" << std::endl;
t.dump();

try {
    t.remove("junk");
} catch (KeyNot FoundException e) {
    std::cerr << "Didn’t find key " << e.key << " in remove." << std::endl << std::endl;
}

t.remove("fourth");
std::cout << "After removing \"fourth\":" << std::endl;
t.dump();


t.remove("seventh");
std::cout << "After removing \"seventh\":" << std::endl;
t.dump();
}
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```
cd cs301
mkdir debugging
cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):

```
cp ~lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

```none
CHTNode.cpp
CHTNode.h
ChainedHashTable.cpp
ChainedHashTable.h
DuplicateKeyException.h
KeyNotFoundException.h
Makefile
tester.cpp
```

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You *could* compile the test program by hand with the following command:

```
g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize this by allowing separate compilation of multiple files to create “object modules” (for C++, these files end in a “.o” extension) which can be linked together to form an executable in a separate stage of compilation. So, you could compile all of the files separately, using the “-c” flag, then link them together at the end:

```
g++ -g -c tester.cpp  
g++ -g -c CHTNode.cpp  
g++ -g -c ChainedHashTable.cpp  
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it changes, both of the .cpp files should be recompiled. Unix provides the make utility to automate this process. The make utility takes a file called “Makefile” as input and automatically figures out which files need to be recompiled based on the rules that Makefile contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging directory. It consists of a sequence of “targets” followed by commands to be executed to create the targets. A line like:

```
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h \  
                    CHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified since ChainedHashTable.o was created. (The “\” is required to indicate that the dependency list spans more than one line.) The following line in the Makefile is the command to execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if you have occasion to modify a Makefile or create one of your own, note that the first character of these command lines MUST be a tab, otherwise you’ll get very strange error messages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in the directory containing the files. When it finishes, type “ls” again. You should now see “.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type

```
ddd <executable-name>
```

where `<executable-name>` is replaced with the name of the executable you want to debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

**Setting a breakpoint**

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

**Examining variable values**

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

   \texttt{t.table[0]@t.size}

into the Argument window (next to the “( ):" label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. Double-click on the key value. This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. Double click on the _M_dataplus field of key. _M_p in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

**Stepping through a program**

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variable. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert(“seventh”, 7) to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump( )” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the the formal parameters have been give for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

**Debugging a “core dump”**

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
ence. Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ run time system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type

   ./tester

   You should see some output from the program followed by the message

   Segmentation Fault (core dumped)

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type

   ddd tester core &

   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-
   Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name man-
   gling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “_ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHT-
   Node. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in
   the source window. This tells us why the segmentation fault occurred.

**Exercise**

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
ddd Quick Reference

Running ddd

ddd <executable-name> where the name of your executable is substituted for <executable-name>

Viewing program output/typing console input

Choose View->Execution window

Setting a breakpoint

Click to the left of the line where you want to stop, then click the “Break” button.

Running the program

Click “Run” in the Command tool or choose Program->Run

Viewing variable values

“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

Viewing the contents of dynamically allocated arrays

Use “@” to specify the length of the array in the Argument area (e.g. a[0]@7)

Following pointers

Double-clicking on a pointer displays what it points to in the Data window.

Single stepping

“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.

“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

Executing to the end of a function/method

To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

Finding the source of a core dump

Type “ddd <executable> core” then choose Status->Backtrace.
// File: CHTNode.h
// Author: Lewis Barnett

// A Node to be used in a Chained Hash Table. It contains a key, which
// is a string, and a data payload, which is an int.
#ifndef __CHTNode__
#define __CHTNode__

#include <string>

class CHTNode {
private:
    std::string key;
    int       data;

public:
    CHTNode *next;

    CHTNode(std::string k, int payload);
    std::string getKey();
    int getData();
};
#endif

#include "CHTNode.h"

// Constructor
CHTNode::CHTNode(std::string k, int payload) {
    key = k;
    data = payload;
}

// Return the key value stored in this node
std::string CHTNode::getKey() {
    return key;
}

// Return the data value stored in this node.
int CHTNode::getData() {
    return data;
}
// File: ChainedHashTable.h
// Author: Lewis Barnett

// A templated Chained Hash table. The keys are assumed to be strings.
// Duplicate keys are not allowed.

#ifndef __ChainedHashTable__
#define __ChainedHashTable__

#include <string>
#include "CHTNode.h"
#include "KeyNameNotFoundException.h"
#include "DuplicateKeyException.h"

// The "declaration" of the class, containing declaration of data members
// and prototypes for methods.
class ChainedHashTable {
  public:
    ChainedHashTable(int slots);
    void insert (std::string k, int value);
    bool lookup (std::string k);
    int retrieve (std::string k);
    void remove(std::string k);
    void dump();

  private:
    // Declaration for a dynamically allocated array of pointers to
    // CHTNodes -- this is the "array of linked lists" that serves
    // as our hash table.
    CHTNode **table;
    // size is the number of slots in table. C++ arrays don’t remember
    // this after they’re created, so we have to save it ourselves.
    int size;

    // Private helper method that generates the hash value for a
    // given key.
    int hash (std::string k);

};

#endif
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include "ChainedHashTable.h"

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // “slots” elements.
    table = new CHTNode* [slots];
    size = slots;
    for (int i = 0; i < size; i++) {
        table[i] = 0; // Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}

return found;

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey())
            break;
    }
    p = p->next;
}
// Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;
        prev->next = p->next;
    }
}
else {
    throw KeyNotFoundException(k);
}

// This is necessary because C++ doesn’t automatically
// garbage collect objects when there are no longer any
// references to them.
delete p;
}

// This is essentially identical to program 14.1 in Sedgewick’s Algorithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
    int h = 0;
    int a = 127;

    for (int i = 0; i < k.length(); i++){
        h = (a * h + k[i]) % size;
    }

    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump(){
    std::cout << “Contents of table:” << std::endl << std::endl;

    for (int i = 0; i < size; i++) {
        std::cout << “Bucket “ << i << “: “ << std::endl;
        CHTNode *p = table[i];
    }
while (p != 0) {
    std::cout << "\tKey = " << p->getKey() << " value = " <<
    p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
}
// tester.cpp
#include <iostream>
#include <string>
#include "CHTNode.h"
#include "ChainedHashTable.h"

main() {  
    // Declare a new hash table that stores ints as its data value.  
    ChainedHashTable t(11);

    // Insert a number of values.  We end up with some chains of
    // length 2.
    t.insert("first", 1);
    t.insert("second", 2);
    t.insert("frist", 32);
    t.insert("third", 3);
    t.insert("fourth", 4);
    t.insert("fifth", 5);
    t.insert("sixth", 6);
    t.insert("seventh", 7);

    // Test the response to inserting a duplicate value.
    try {
        t.insert("third", 9);
    } catch (DuplicateKeyException e) {
        // std::cerr is a separate output stream used for error reporting.
        // Even if the standard output stream is redirected to a file,
        // std::cerr's output appears on the screen.
        std::cerr << "Attempted to insert duplicate key " << e.key
        << std::endl;
    }

    t.dump();

    int val = t.retrieve("sixth");
    std::cout << "retrieve("sixth") returns " << val << std::endl;

    try {
        t.retrieve("bungle");
    } catch (KeyNotFoundException e) {
        std::cerr << "Didn't find key " << e.key << " in retrieve." << std::endl;
    }

    t.remove("sixth");

    std::cout << "After removing "sixth":" << std::endl;
t.dump();

try {
    t.remove("junk");
} catch (KeyNotFoundException e) {
    std::cerr << "Didn't find key " << e.key << " in remove." << std::endl << std::endl;
}

t.remove("fourth");
std::cout << "After removing \"fourth\":" << std::endl;
t.dump();

t.remove("seventh");
std::cout << "After removing \"seventh\":" << std::endl;
t.dump();
}
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```
cd cs301
mkdir debugging
cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you'll get an error if you don’t include it):

```
cp ~/lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

```
CHTNode.cpp
CHTNode.h
ChainedHashTable.cpp
ChainedHashTable.h
DuplicateKeyException.h
KeyNotFoundException.h
Makefile
tester.cpp
```

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You *could* compile the test program by hand with the following command:

```
g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize this by allowing separate compilation of multiple files to create “object modules” (for C++, these files end in a “.o” extension) which can be linked together to form an executable in a separate stage of compilation. So, you could compile all of the files separately, using the “-c” flag, then link them together at the end:

```
g++ -g -c tester.cpp
g++ -g -c CHTNode.cpp
g++ -g -c ChainedHashTable.cpp
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it changes, both of the .cpp files should be recompiled. Unix provides the `make` utility to automate this process. The make utility takes a file called “Makefile” as input and automatically figures out which files need to be recompiled based on the rules that Makefile contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging directory. It consists of a sequence of “targets” followed by commands to be executed to create the targets. A line like:

```
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h \ 
   CHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified since ChainedHashTable.o was created. (The “\” is required to indicate that the dependency list spans more than one line.) The following line in the Makefile is the command to execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if you have occasion to modify a Makefile or create one of your own, note that the first character of these command lines MUST be a tab, otherwise you’ll get very strange error messages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in the directory containing the files. When it finishes, type “ls” again. You should now see “.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type

   `ddd <executable-name>`

   where `<executable-name>` is replaced with the name of the executable you want to debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you’ll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

**Setting a breakpoint**

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

**Examining variable values**

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
Debugging using the ddd debugger

object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make sense to dereference a direct reference.) At this point, it will be a good idea to resize the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the right end of the divider between the Data and Source windows. You should see a box in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this will be a little difficult because it is a dynamically allocated array. The number of elements in the array is given by the size data member, so we can tell the debugger how big it is. Type

   \[ t\text{.table}[0]@t\text{.size} \]

   into the Argument window (next to the “( )” label in the upper left-hand corner) and then click on the “Display” button in the toolbar.

   At this point, you should see another box in the Data area representing the “table” array. The contents of this array are the pointers to the linked lists for each “bucket” in our hash table. The values are shown in hexadecimal notation, indicated by the leading “0x.” The value 0x0 is the null pointer.

Following pointers

One of the big advantages of a debugger is that it lets you easily follow your pointers around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the Data area showing the CHTNode at the head of the list for the 0th bucket in the table. Notice that there is a line in this box for each data member of the CHTNode, the key (a string), the data (an int) and the next link (a pointer to another CHTNode). Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one node. Also notice that the value of key is reported as {...}. This indicates that this data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should now see a label on the arrow pointing from the table box to the CHTNode box which reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of table.

3. Double-click on the key value. This expands the string object (it was a direct reference, not a pointer to a string instance) and shows you the internal structure of the string class, which is rather complicated.

4. Double click on the \_M\_dataplus field of key. \_M\_p in the resulting display box is a pointer to the string’s contents. Double click on it to see the value contained in this string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

**Stepping through a program**

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variable. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert(“seventh”, 7) to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump( )” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the formal parameters have been given for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

**Debugging a “core dump”**

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
ence. Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ run time system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type
   .tester
   You should see some output from the program followed by the message
   Segmentation Fault (core dumped)

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type
   ddd tester core &
   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name mangling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHTNode. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

Exercise

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.
**ddd Quick Reference**

**Running ddd**

`ddd <executable-name>` where the name of your executable is substituted for `<executable-name>`

**Viewing program output/typing console input**

Choose View->Execution window

**Setting a breakpoint**

Click to the left of the line where you want to stop, then click the “Break” button.

**Running the program**

Click “Run” in the Command tool or choose Program->Run

**Viewing variable values**

“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

**Viewing the contents of dynamically allocated arrays**

Use “@” to specify the length of the array in the Argument area (e.g. `a[0]@7`)

**Following pointers**

Double-clicking on a pointer displays what it points to in the Data window.

**Single stepping**

“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.

“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

**Executing to the end of a function/method**

To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

**Finding the source of a core dump**

Type “`ddd <executable> core`” then choose Status->Backtrace.
class CHTNode {
    private:
    std::string key;
    int data;

    public:
    CHTNode *next;

    CHTNode(std::string k, int payload);
    std::string getKey();
    int getData();
};

}
// A templated Chained Hash table. The keys are assumed to be strings. // Duplicate keys are not allowed.

#ifndef __ChainedHashTable__
#define __ChainedHashTable__

#include <string>
#include "CHTNode.h"
#include "KeyNotFoundException.h"
#include "DuplicateKeyException.h"

// The “declaration” of the class, containing declaration of data members 
// and prototypes for methods.
class ChainedHashTable {
 public:
   ChainedHashTable(int slots);
   void insert (std::string k, int value);
   bool lookup (std::string k);
   int retrieve (std::string k);
   void remove (std::string k);
   void dump();

double size; // size is the number of slots in table. C++ arrays don’t remember 
// this after they’re created, so we have to save it ourselves.

 private:
   // Declaration for a dynamically allocated array of pointers to 
   // CHTNodes -- this is the “array of linked lists” that serves 
   // as our hash table.
   CHTNode **table;

   // Private helper method that generates the hash value for a 
   // given key.
   int hash (std::string k);
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include “ChainedHashTable.h”

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // “slots” elements.
    table = new CHTNode* [slots];
    size = slots;

    for (int i = 0; i < size; i++) {
        table[i] = 0;// Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}

return found;

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn’t contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey()) {
            found = true;
            break;
        }
        p = p->next;
    }
// Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;

        prev->next = p->next;

        // This is necessary because C++ doesn’t automatically
        // garbage collect objects when there are no longer any
        // references to them.
        delete p;
    }
} else {
    throw KeyNotFoundException(k);
}

// This is essentially identical to program 14.1 in Sedgewick’s Algo-
// rithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
    int h = 0;
    int a = 127;

    for (int i = 0; i < k.length(); i++){
        h = (a * h + k[i]) % size;
    }

    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump(){
    std::cout << “Contents of table:” << std::endl << std::endl;

    for (int i = 0; i < size; i++) {
        std::cout << “Bucket “ << i << “: ” << std::endl;
        CHTNode *p = table[i];
    }
while (p != 0) {
    std::cout << "\tKey = " << p->getKey() << " value = " <<
    p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
}
// tester.cpp
#include <iostream>
#include <string>
#include "CHTNode.h"
#include "ChainedHashTable.h"

main() {
    // Declare a new hash table that stores ints as its data value.
    ChainedHashTable t(11);

    // Insert a number of values. We end up with some chains of
    // length 2.
    t.insert("first", 1);
    t.insert("second", 2);
    t.insert("frist", 32);
    t.insert("third", 3);
    t.insert("fourth", 4);
    t.insert("fifth", 5);
    t.insert("sixth", 6);
    t.insert("seventh", 7);

    // Test the response to inserting a duplicate value.
    try {
        t.insert("third", 9);
    } catch (DuplicateKeyException e) {
        // std::cerr is a separate output stream used for error reporting.
        // Even if the standard output stream is redirected to a file,
        // std::cerr's output appears on the screen.
        std::cerr << "Attempted to insert duplicate key " << e.key
                  << std::endl;
    }

    t.dump();

    int val = t.retrieve("sixth");
    std::cout << "retrieve("sixth") returns " << val << std::endl;

    try {
        t.retrieve("bungle");
    } catch (KeyNotFoundException e) {
        std::cerr << "Didn't find key " << e.key << " in retrieve." << std::endl;
    }

    t.remove("sixth");

    std::cout << "After removing "sixth":" << std::endl;"
t.dump();

try {
    t.remove("junk");
} catch (KeyNotfoundException e) {
    std::cerr << "Didn't find key " << e.key << " in remove." <<
              std::endl << std::endl;
}

t.remove("fourth");
std::cout << "After removing ""fourth"":" << std::endl;
t.dump();


t.remove("seventh");
std::cout << "After removing ""seventh"":" << std::endl;
t.dump();
}
Debugging using the ddd debugger

Introduction

In general debugging is the process of finding runtime errors or “bugs” in a program. Usually a compiled program is run with carefully selected data and if the output is not what is expected, then the programmer must engage in debugging to find the source of the error.

One simple way to discover errors is to insert output statements in your code that will show the progress of the program and display the value of key variables. This is a tried and true method that works under essentially any circumstances, and is sufficient to determine the cause of many problems. It has the drawback that output is not always sent immediately to the screen due to internal buffering. This has the practical effect that if an error occurs, the output may not appear on the screen until several subsequent instructions been executed. If an error causes the termination of the program before this occurs, output may not show on the screen and the programmer is unable to determine where the error is. This is partially solved by using “flush” methods which force output immediately to the screen.

While using output statements works in simple cases it often is not efficient when runtime exceptions occur in the code and it is not clear in which methods the exceptions are being generated. In a complicated sequence of method calls it becomes tedious to discover the source of the problem.

Debuggers are tools that allow controlled execution of a program. During execution the user can have the execution pause at specified lines in the program called breakpoints. At a breakpoint the user can see values of variables and can reset values of variables. A debugger also allows the user to have execution progress one instruction at a time - called “single stepping”.

By carefully setting breakpoints the user of a debugger can quickly find the part of the program where an error is occurring.

You may be familiar with the debugger in IDEs you have used in earlier courses. Most of the debugging tools available under Unix are “command line” debuggers (e.g. dbx and gdb), but several graphical front ends have been written which behave much like the debuggers in windows based IDEs. One such front end is called “ddd,” and it is available on all of our Unix systems.

Using ddd

This exercise will demonstrate the basic operation of ddd by walking you through a debugging session on an example program. To begin, you should log in to a Unix system, either at the console of one of the Sparcstations in G30, or by using Exceed from one of the PCs in either G30 or G25. (If you have installed a X server package on your own computer, you can use that setup as well.) Once you are logged in, start up a terminal window
by clicking on the icon resembling a computer at the left end of the front panel.

**Retrieving the files**

The files for this exercise are in my Unix account. Before retrieving them, change into your cs301 directory and create a new folder for the files, then change into the new directory:

```bash
cd cs301
mkdir debugging
cd debugging
```

When you have completed these commands, type “pwd” (without the quotes - this stands for “print working directory”) to be sure you are in the right place.

Now, type the following command (note the “.” at the end -- you’ll get an error if you don’t include it):

```bash
cp ~/lbarnett/share/debugging/* .
```

The “*” is a wildcard character which indicates that every file in the directory “debugging” should be copied.

Now, type “ls” to see the files in your debugging directory. You should see these files:

- CHTNode.cpp
- CHTNode.h
- ChainedHashTable.cpp
- ChainedHashTable.h
- DuplicateKeyException.h
- KeyNotFoundException.h
- Makefile
- tester.cpp

The source code for these files (except for the exception classes and the Makefile) are attached to the end of this handout. These files implement a chained hash table that stores integer values with keys that are strings. “Makefile” is an input file to a program maintenance utility program. It describes how to compile the executable for tester, which is a short test program for the hash table.

Have a brief look at CHTNode.h and ChainedHashTable.h to familiarize yourself with the data structures the classes use. CHTNode is a singly linked list node with a string for the key and an int for the data. ChainedHashTable has an int representing the size of the hash table and a dynamically allocated array of pointers to CHTNodes which serves as the table itself. We will refer to the elements of this array as “buckets.”

**Compiling the test program**

You *could* compile the test program by hand with the following command:

```bash
g++ -g tester.cpp ChainedHashTable.cpp CHTNode.cpp -o tester
```
If you were still working on the program and needed to recompile frequently, this would end up wasting a lot of CPU cycles, because changes to CHTNode.cpp, for example, mean that only CHTNode.cpp needs to be recompiled. C++ and many other languages recognize this by allowing separate compilation of multiple files to create “object modules” (for C++, these files end in a “.o” extension) which can be linked together to form an executable in a separate stage of compilation. So, you could compile all of the files separately, using the “-c” flag, then link them together at the end:

```
g++ -g -c tester.cpp
g++ -g -c CHTNode.cpp
g++ -g -c ChainedHashTable.cpp
```

```
g++ tester.o CHTNode.o ChainedHashTable.o -o tester
```

Now, if you change CHTNode.cpp, you only have to recompile that file, then repeat the final link stage.

For large programs, it gets hard to keep track of when you have to recompile things. For example, both CHTNode.cpp and ChainedHashTable.cpp use CHTNode.h, so whenever it changes, both of the .cpp files should be recompiled. Unix provides the `make` utility to automate this process. The make utility takes a file called “Makefile” as input and automatically figures out which files need to be recompiled based on the rules that Makefile contains.

Use xemacs to have a look at the Makefile you’ve just copied into your debugging directory. It consists of a sequence of “targets” followed by commands to be executed to create the targets. A line like:

```
ChainedHashTable.o: ChainedHashTable.cpp ChainedHashTable.h \
  CHTNode.h
```

indicates that the “target” ChainedHashTable.o should be recompiled if any of the three files ChainedHashTable.cpp, ChainedHashTable.h or CHTNode.h have been modified since ChainedHashTable.o was created. (The “\” is required to indicate that the dependency list spans more than one line.) The following line in the Makefile is the command to execute to create the updated version of ChainedHashTable.o. (IMPORTANT NOTE: if you have occasion to modify a Makefile or create one of your own, note that the first character of these command lines MUST be a tab, otherwise you’ll get very strange error messages.)

To create an executable to try out ddd upon, simply type “make” (without the quotes) in the directory containing the files. When it finishes, type “ls” again. You should now see “.o” files for each of the “.cpp” files, plus a file called “tester,” which is the executable we’ve created.

**Running the debugger**

1. In the directory where your source and executables are, type

```
ddd <executable-name>
```

where `<executable-name>` is replaced with the name of the executable you want to debug. For this exercise, type
After a few seconds, you should see a window with the source code for tester.cpp. The panel below the source code is a window where you can interact directly with gdb (Gnu Debugger) if you so choose. Just about everything you'll be interested in doing has a menu item, so this window is not going to be terribly important, but if you want to debug over a telnet connection, learning about the command language of gdb will be necessary. Any console output your program produces will also appear in this window.

2. From the “View” menu, choose “Execution window.” This will pop up a separate window where you can interact with your program. This is a bit less confusing than trying to use the gdb window to view the output or provide keyboard input for your program.

Recall that tester is a test driver for a simple chained hash table class. It inserts a few (key, value) pairs to the table, dumps out the contents, tests for a couple of exceptional conditions, deletes a few values, then dumps the table again.

Setting a breakpoint

Before we execute the program, we need to set a breakpoint so that the program won’t just execute all the way through to termination.

1. Left-click the mouse to the left of the line where “seventh” is inserted into the table. You should see the value “tester.cpp:19” appear in the small “argument window” in the upper left-hand corner.

2. Click on the “Break” button in the toolbar. A small red Stop sign should appear in the source listing.

3. Choose “Run” from the “Program” menu. A dialog box will pop up asking for any arguments you want to supply. Our program doesn’t expect any command line arguments, so you can just click the “Run” button. Some status messages will appear in the gdb window, and a green arrow will appear to the left of the breakpoint icon in the source code, indicating that execution has halted at the breakpoint.

Examining variable values

1. Move the mouse over the variable name “t” on line 19. The value of t will appear in the status bar at the bottom of the window. A “tooltip” box will also pop up showing some information about t. These are the values of the two data members of t, the array of pointers to linked lists, and the size data member. This is nice, but it doesn’t actually tell us too much. It would, however, be very helpful for variables of primitive types.

2. Right-click on t and select “Display t” from the resulting pop-up menu. This causes the Data window to appear above the source window. (Note that you had the choice of either displaying t or *t in the menu. Since t is an actual ChainedHashTable
object, not a pointer to a ChainedHashTable, we chose the former. It doesn’t make
sense to dereference a direct reference.) At this point, it will be a good idea to resize
the window and adjust things so that more of the Data Window is visible.

3. Resize the window by dragging a section of the window border. This works more or
less like it does in Windows.

4. Adjust the amount of space the Data window has by dragging the small square at the
right end of the divider between the Data and Source windows. You should see a box
in the Data window labelled “t” and showing the values of the two data members.

5. We’d like to see the values of all of the elements of the table data member, but this
will be a little difficult because it is a dynamically allocated array. The number of
elements in the array is given by the size data member, so we can tell the debugger
how big it is. Type

\[
t.table[0]@t.size
\]

into the Argument window (next to the “( )” label in the upper left-hand corner) and
then click on the “Display” button in the toolbar.

At this point, you should see another box in the Data area representing the “table”
array. The contents of this array are the pointers to the linked lists for each “bucket”
in our hash table. The values are shown in hexadecimal notation, indicated by the
leading “0x.” The value 0x0 is the null pointer.

**Following pointers**

One of the big advantages of a debugger is that it lets you easily follow your pointers
around to see when you’ve gotten things linked up wrong.

1. Double-click on the 0th element of the table array. A new box should pop up in the
Data area showing the CHTNode at the head of the list for the 0th bucket in the
table. Notice that there is a line in this box for each data member of the CHTNode,
the key (a string), the data (an int) and the next link (a pointer to another CHTNode).
Notice that next is 0x0 (the null pointer), so this bucket’s chain contains only one
node. Also notice that the value of key is reported as {...}. This indicates that this
data member is actually a class instance (the string class).

2. Use the mouse to drag this box up and away from the table display box. You should
now see a label on the arrow pointing from the table box to the CHTNode box which
reads “*( )[0]” – indicating that this is the dereferenced value of the 0th element of

3. Double-click on the key value. This expands the string object (it was a direct refer-
ence, not a pointer to a string instance) and shows you the internal structure of the
string class, which is rather complicated.

4. Double click on the _M_dataplus field of key. _M_p in the resulting display box is a
pointer to the string’s contents. Double click on it to see the value contained in this
string.
5. Now, double-click the next-to-last element of the table array, and expand the key field as before to see the value of the key. (You may see “fifthh” rather than “fifth.” I believe this to be a bug in ddd, not in the program you are examining.) Notice that the next field of this node is not null.

6. Double-click on the next field of the node you have just displayed. This should pop up another CHTNode instance. Expand the key field as before to see the value of the key.

**Stepping through a program**

The other big advantage conveyed by debuggers is the ability to “single step” through a program. This means executing one line of the program at a time, allowing you to see the effects of the execution immediately by examining the resulting values of variable. There are two flavors of single stepping provided by most debuggers, including ddd. The “step” command advances execution by one source line, “stepping into” function or method calls when they are encountered. The “next” command “steps over” function calls, executing them in their entirety and stopping again at the next source line in the calling program.

1. In the “View” menu, choose “Command tool.” This will pop up a small palette containing buttons for most of the execution commands.

2. Click on the “Next” button in the Command tool. This causes the line “t.insert(“seventh”, 7) to be executed. Notice that one of the elements in the table display is now highlighted, indicating that this value changed as a result of this step. The green arrow is now on the next executable statement in the program.

3. Set a breakpoint on the line where “t.dump( )” is called. (This will cause the Command tool to disappear, so choose View->Command Tool to get it back again.)

4. Let’s try stepping into a function. Click on the “Step” button in the command tool. We should end up in the definition of the ChainedHashTable insert method.

5. In the “Data” menu, choose “Display arguments.” This shows you the values of the the formal parameters have been give for this call to insert.

6. Let the mouse pointer come to rest over the “k” and the “value” in the parameter list of insert to demonstrate another way of seeing these values.

7. This call to insert will throw an exception because the key “third” is already in the table. Call up the Command Tool again. Click the “Cont” button. This will cause the execution of the program to “continue” until the next breakpoint (or the end of the program) is encountered.

8. Exit ddd now by choosing “Exit” from the “File” menu.

**Debugging a “core dump”**

In the coming weeks, you will become quite familiar with the error message “Segmentation fault: core dumped.” This message can mean one of a number of things, but it always results when your program attempts to access a memory location outside its allocated “segment.” This could happen if, for example, you used an invalid index in an array refer-
Another way to generate this error is by trying to dereference a null pointer, which will try to access the reserved memory location 0x0.

The ChainedHashTable class we are using for demonstration purposes has a bug that results in a segmentation fault. When this occurs, the C++ runtime system produces a “core dump.” “Core” in this context is a terminology hangover from the days when main memory was constructed of magnetic cores; so a “core dump” is a complete image of the memory state of the executing program when the fatal error occurred. This can be quite handy for debugging purposes, if you have the proper tools to exploit it.

1. In your terminal window, type

   ./tester

   You should see some output from the program followed by the message

   Segmentation Fault (core dumped)

2. Type “ls” to view the contents of your directory. You should see a file called “core.”

3. Type

   ddd tester core &

   in your terminal window. Once ddd starts, you should see a message in the debugger window that indicates where the fatal error occurred; in this case, it was in the get-Data method of the CHTNode class. (You will actually see something different due to the way g++ compiles templated classes. It uses a technique called “name mangling” which adds additional information to the names of methods based on what data type you’ve used for the template. for example, what I actually saw for the method name was “_ZN7CHTNode7getDataEv.”) Just knowing what method the error occurred in, however, isn’t always terribly helpful, since this method is called from several different places in ChainedHashTable.

4. Choose “Backtrace” from the “Status” menu. This should pop up a dialog box that displays the program’s call stack at the time of the error. This is the sequence of function calls that were currently active when the program terminated. From this you can see that main( ) reached line 34 in tester.cpp, whereupon it called retrieve( ) in ChainedHashTable. At line 72 in retrieve( ), a call was made to getData( ) in CHT-Node. This tells us exactly where things went wrong.

5. Double-click on the line for the call to retrieve( ) in the Backtrace window. This should show you the line in the source window where the call that the line describes occurred in your source code. Now, hover your mouse pointer over the variable p in the source window. This tells us why the segmentation fault occurred.

**Exercise**

The retrieve method in ChainedHashTable contains two bugs. Use ddd to discover what they are and fix them. Print out a copy of your modified ChainedHashTable.cpp to turn in at our next class meeting.


**ddd Quick Reference**

**Running ddd**

ddd `<executable-name>` where the name of your executable is substituted for `<executable-name>`

**Viewing program output/typing console input**

Choose View->Execution window

**Setting a breakpoint**

Click to the left of the line where you want to stop, then click the “Break” button.

**Running the program**

Click “Run” in the Command tool or choose Program->Run

**Viewing variable values**

“Hover” the mouse over the variable name in the source window, or double-click on it to see its value in the Data window.

**Viewing the contents of dynamically allocated arrays**

Use “@” to specify the length of the array in the Argument area (e.g. `a[0]@7`)

**Following pointers**

Double-clicking on a pointer displays what it points to in the Data window.

**Single stepping**

“Step” executes one line of source code. If the line contains a function/method call, execution stops at the first line of the called function.

“Next” executes one line of source code. If the line contains a function/method call, the call is executed in its entirety, its return value (if any) is displayed in the debugger window, and execution stops on the source line after the one containing the call.

**Executing to the end of a function/method**

To complete execution of a method or function and halt on the line following the call to the method, click the “Finish” button on the Command tool. (May not work well for functions that throw exceptions.)

**Finding the source of a core dump**

Type “ddd `<executable>` core” then choose Status->Backtrace.
class CHTNode {
private:
    std::string key;
    int data;

public:
    CHTNode *next;

    CHTNode(std::string k, int payload);
    std::string getKey();
    int getData();
};
A templated Chained Hash table. The keys are assumed to be strings. Duplicate keys are not allowed.

```cpp
#ifndef __ChainedHashTable__
define __ChainedHashTable__

#include <string>
#include "CHTNode.h"
#include "KeyNotFoundException.h"
#include "DuplicateKeyException.h"

// The "declaration" of the class, containing declaration of data members
// and prototypes for methods.
class ChainedHashTable {
 public:
  ChainedHashTable(int slots);
  void insert (std::string k, int value);
  bool lookup (std::string k);
  int retrieve (std::string k);
  void remove(std::string k);
  void dump();

 private:
  // Declaration for a dynamically allocated array of pointers to
  // CHTNodes -- this is the "array of linked lists" that serves
  // as our hash table.
  CHTNode **table;
  // size is the number of slots in table. C++ arrays don’t remember
  // this after they’re created, so we have to save it ourselves.
  int   size;

  // Private helper method that generates the hash value for a
  // given key.
  int   hash (std::string k);

};

#endif
```
// File: ChainedHashTable.cpp
// Author: Lewis Barnett

// The “implementation” of the ChainedHashTable class, i.e. all of the
// method definitions.
#include <iostream>
#include "ChainedHashTable.h"

// Constructor - allocates the table and initializes all slots to
// null (0).
ChainedHashTable::ChainedHashTable(int slots){
    // This says: create an array of pointers to CHTNodes with
    // “slots” elements.
    table = new CHTNode* [slots];
    size = slots;

    for (int i = 0; i < size; i++) {
        table[i] = 0;// Make each element of table a null pointer.
    }
}

// Insert a new (key, value) pair into the table.
// If the key k has previously been inserted into the table, a
// DuplicateKeyException is thrown.
void ChainedHashTable::insert(std::string k, int value){
    if (lookup(k)) {
        // Note that we don’t have to create the exception object with
        // new as we did in Java.
        throw DuplicateKeyException(k);
    }

    int bucket = hash(k);
    CHTNode *n = new CHTNode(k, value);

    // Insert the new (key, value) pair at the beginning of the chain
    // for this bucket.
    n->next = table[bucket];
    table[bucket] = n;
}

// Discover whether a key is currently stored in the table.
bool ChainedHashTable::lookup (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];

    return found;
}
while (!found && p != 0) {
    if (k == p->getKey())
        found = true;
    p = p->next;
}

return found;
}

// Return the data associated with a key which is currently stored in
// the table. Throws a KeyNotFoundException if the key isn't contained
// within the table.
int ChainedHashTable::retrieve (std::string k) {
    bool found = false;
    int bucket = hash(k);

    CHTNode *p = table[bucket];
    while (!found) {
        if (k == p->getKey())
            found = true;
        p = p->next;
    }

    if (found) {
        return p->getData();
    } else {
        throw KeyNotFoundException(k);
    }
}

// Remove the data associated with the specified key from the table.
// If the key is not found, throw a KeyNotFoundException.
void ChainedHashTable::remove (std::string k) {
    // First, find the position of the requested key
    bool found = false;
    int bucket = hash(k);

    // Scan the list for the correct bucket to find the key
    CHTNode *p = table[bucket];
    while (!found && p != 0) {
        if (k == p->getKey()) {
            found = true;
            break;
        }
        p = p->next;
    }
}
// Now, we need a pointer to the preceding node in the list
if (found) {
    // Deleting the first key in the bucket is a special case
    if (p == table[bucket]) {
        table[bucket] = p->next;
    } else {
        // We scan the list a second time here; a more efficient
        // implementation would keep track of a trailing pointer
        // in the initial loop.
        CHTNode *prev = table[bucket];
        while (prev->next != 0 && prev->next != p)
            prev = prev->next;

        prev->next = p->next;

        // This is necessary because C++ doesn’t automatically
        // garbage collect objects when there are no longer any
        // references to them.
        delete p;
    }
} else {
    throw KeyNotFoundException(k);
}
}

// This is essentially identical to program 14.1 in Sedgewick’s Algo-
// rithms
// in C++, 3e.
int ChainedHashTable::hash(std::string k){
    int h = 0;
    int a = 127;

    for (int i = 0; i < k.length(); i++){
        h = (a * h + k[i]) % size;
    }

    return h;
}

// A utility method to dump out the table in a readable form.
void ChainedHashTable::dump(){
    std::cout << “Contents of table:” << std::endl;

    for (int i = 0; i < size; i++) {
        std::cout << “Bucket “ << i << “: “ << std::endl;
        CHTNode *p = table[i];
while (p != 0) {
    std::cout << \"\tKey = \" << p->getKey() << \" value = \" <<
          p->getData() << std::endl;
    p = p->next;
}
std::cout << std::endl;
}
// tester.cpp
#include <iostream>
#include <string>
#include "CHTNode.h"
#include "ChainedHashTable.h"

main() {
    // Declare a new hash table that stores ints as its data value.
    ChainedHashTable t(11);

    // Insert a number of values. We end up with some chains of
    // length 2.
    t.insert("first", 1);
    t.insert("second", 2);
    t.insert("frist", 32);
    t.insert("third", 3);
    t.insert("fourth", 4);
    t.insert("fifth", 5);
    t.insert("sixth", 6);
    t.insert("seventh", 7);

    // Test the response to inserting a duplicate value.
    try {
        t.insert("third", 9);
    } catch (DuplicateKeyException e) {
        // std::cerr is a separate output stream used for error reporting.
        // Even if the standard output stream is redirected to a file,
        // std::cerr's output appears on the screen.
        std::cerr << "Attempted to insert duplicate key " << e.key
                   << std::endl;
    }

    t.dump();

    int val = t.retrieve("sixth");
    std::cout << "retrieve("sixth") returns " << val << std::endl;

    try {
        t.retrieve("bungle");
    } catch (KeyNotFoundException e) {
        std::cerr << "Didn’t find key " << e.key << " in retrieve." <<
                   std::endl << std::endl;
    }

    t.remove("sixth");

    std::cout << "After removing "sixth":" << std::endl;
}
t.dump();

try {
    t.remove("junk");
    catch (KeyNotFound & e) {
        std::cerr << "Didn’t find key " << e.key << " in remove." << std::endl << std::endl;
    }
}

    t.remove("fourth");
    std::cout << "After removing ""fourth\":" << std::endl;
    t.dump();

    t.remove("seventh");
    std::cout << "After removing ""seventh\":" << std::endl;
    t.dump();
}