CSI 333 – Programming at the Hardware/Software Interface – Fall 1999
Laboratory Exercise 5

INSTRUCTIONS: Copy `~csi333/Lab5/WAdemo.cpp` and `~csi333/Lab5/WAdemo.h` into a new directory. Compile and link `WAdemo.cpp` with debugging symbols (use the `-g` option). Run it under the emacs “gud” interface to `gdb` as described in Lab 4. Follow the instructions below.

Write notes on this sheet where questions are asked and when you see something “interesting.” Ask the TAs or your neighbors for any help or explanation.

Sometime DURING the lab (BEFORE the last 1/2 hour) show the TA that you have made some progress to get the passing checkoff. You MUST attend a lab session and demonstrate some skill with using the debugger commands below in order to get passing credit. If you must miss your lab session, just attend another.

For the final lab 5 grade, complete answering the questions and complete converting all the plain functions and code to exercise them to member functions. Append a paper listing of your code (.cpp file and .h file) and get it to your TA within 2 weeks of your lab session.

Before or after the lab session, study the last sheet and tear it off so you can keep it for studying for the final exam.

1 Set Breakpoints

(gdb) list

(gdb) break main

(gdb) break WAListInit

(gdb) break WAListPutFront

(gdb) break WAList::PutFront

(gdb) break WAListPutEnd

(gdb) run
2  Observe Type and Value of an Automatic Variable

(gdb) ptype mylist

(gdb) print mylist

(gdb) display mylist

3  Observe a Function stopped by a Breakpoint

After the "continue" command below, the program will stop in WAListInit because you gave the "break WAListInit" above.
(gdb) continue

4  Observe how Struct Fields Change on Assignments

(gdb) display *pWAList

(gdb) next

(gdb) next

5  Continue and Stop at the Next Breakpoint

(gdb) contin

You should be in WAList::PutFront.

6  Study Variables Available in a Member Function Body

(gdb) print this
(gdb) print *this

(gdb) whatis this

(gdb) whatis *this

(gdb) display *this

(gdb) whatis first

(gdb) whatis last

(gdb) display first

(gdb) display last

(gdb) next
Command next a few times until the fields of *this become non-zero.

(gdb) print *first

(gdb) print *last

(gdb) continue

7 Observe how a Link is Appended to a Linked List

Program should be stopped at the beginning of \texttt{WAListPutEnd}

(gdb) display *pWAList
(gdb) display *(pWAList->first)
(gdb) display *(pWAList->last)
(gdb) next
Command next a few times until the "First" and "Second" lines are visible.

(gdb) print *pWAList
(gdb) print *pWAList->first
(gdb) print *pWAList->first->next
(gdb) print *pWAList->first->next->next
Tip for using emacs *gud* interface: Position the cursor on a command already typed, edit it, then press enter.

8 Observing Program's final Result

(gdb) finish
Command a few times until control goes back to main.
Give continue and finish commands until main gets to its while loop.

(gdb) print mylist
(gdb) print mylist.first
(gdb) print *(mylist.first)
(gdb) print *(mylist.first->next)
(gdb) print *(mylist.first->next->next)
(gdb) print *(mylist.first->next->next->next)

9 Observing Addresses of Function (Code) Bodies

Write down here the Hexadecimal numeric addresses that are reported by the commands:
(gdb) print &main

(gdb) print &WALInit

(gdb) print &WALPutFront

(gdb) print &WAL::PutFront
10 Quit

Give the debugger the quit command.

11 Observe the Compiler’s Output of Assembly Language

Before or after your lab session, read the material about assemblers at the end of this handout.

After you copy the *.cpp and *.h files into a directory where you have write permission, run the compiler driver command and directory list commands:

```
g++ -S WAdemo.cpp
ls
```

Observe the file WAdemo.s Examine its contents with a text editor or the less command.

11.1 Finding Labels

Find the Labels that begin with the names below: (Remember, a label denoted by a name followed by the colon “;” character.) You will see that after each of these substrings, there will be an underscore character followed by messy looking letters and digits.

```
main
```

```
WALListInit
```

```
WALListPutFront
```

```
PutFront
```

Under each label prefix, copy the first 7 lines of assembly language.
11.2 Finding Function Calls

Examine the assembly language code that follows the label main:
Look for each call instruction and observe its operand. Each operand is a symbol: The same sort of symbol that you used in MIPS assembly language programming.

Try to figure out which line of C++ source was compiled to produce each function call in the assembly language listing.

12 Observing Symbols from the Executable File’s Symbol Table

Compile and link WAdemo.cpp with the command
g++ -o WAdemo WAdemo.cpp

Then, examine the output of the each of the two commands
(1) nm WAdemo
(2) nm --demangle WAdemo

Write copy the report lines from these symbol table printing command that display the name and values of the symbols for the four function bodies you recorded above. Compare the symbol values with the values you wrote down during the debugger session.

13 Complete Coding the Operations on WAList as Member Functions

Do this and revise main to use member functions instead of plain functions.

14 Writing a Constructor

Write a function body into WAdemo.cpp that begins:

WAList::WAList( void )
{
    cout << "Constructor for WAList is Running.." << endl;
    // Put code equivalent to that inside WAListInit( void )
    // here
}

Uncomment the declaration of the constructor inside the declaration of struct WAList. Recompile and run WAdemo. Try to figure out when WAList::WAList() was called. What is an advantage of initializing a data structure instance with an initializer over initializing it with an ordinary initialization function? (Hint: Programmers are sometimes forgetful.)
15 About Assemblers

An assembler has basically two functions:

1. The first function is to translate the mnemonics and their operands on each line into machine instructions or data and arrange for these results to eventually get loaded into a computer memory. Some mnemonics denote assembler control operations, such as selecting the destination segment for the code that follows or just reserving space. This function requires the assembler to determine the memory address for each translated item, in addition to translating the mnemonic instruction name, operands and data values into bits. Thus an assembly language programmer (or compiler software) does not have to figure out the numerical addresses where machine instructions and memory data will be located. The SPIM assembler determined the absolute (or final) address for each instruction and data word it assembles. Real assemblers generally determine relative addresses which are offsets from the beginning of a section being assembled.

2. The second function is to support symbols which symbolize addresses and which can be used for operands, so that the numeric address for a variable or block of machine code does not have to be known in order to write code that refers to it. An assembler will typically process the input file sequentially. It will determine the address for each instruction or data item when it processes its mnemonic and operands the first time. A symbol becomes defined when the symbol appears as a label: When the label is encountered, the assembler will determine the numeric address that it symbolizes. (Some real assemblers have certain statements whose purpose is to define or set properties of symbols; the syntax rules of such statements usually make the symbol an operand.)

A forward reference is the appearance of a symbol as an operand somewhere in the input file before that symbol appears as a label (and before it is otherwise defined.) Since the instruction or data mnemonic cannot be completely assembled until the value that the symbol symbolizes is known, forward references pose a problem for assemblers.

To support symbols, the assembler will maintain a symbol table that records the name (a character string) and all relevant information about a symbol whenever the information becomes known. In particular, the symbol table entry indicates whether or not the symbol is defined. For every symbol that is defined, the symbol table entry will record the value of the symbol.

The easiest way to solve the forward reference problem is to make the assembler take two passes through the input file. The first pass determines the address of every instruction or data item to be assembled, accumulates the definition (i.e., address symbolized by) of each defined symbol, and determines which symbols remain undefined.

The second pass uses the numeric definitions in the symbol table to assemble numeric addresses for symbolic operands. The second pass does all the necessary translations, composes bit strings representing machine instructions out of their constituent fields, and takes care of outputting the results.

Real assemblers permit symbols to remain undefined. They emit into the object file the symbol table plus a table that indicates all the places in the code or data affected by each undefined symbol. A linker uses these tables to resolve each reference to an undefined symbol in one object file with a definition from another object file. Today, dynamic linking is popular: Undefined symbols in
executable files are resolved by objects in shared library archive files when the executable program is run.)

Patterson and Hennessy give an introduction to assembling, linking and loading in Section (3.9), which is required reading.