CSCI 463 Assignment 3 – Memory Simulator

20 Points

Abstract

In this assignment, you will write a C++ program to simulate a computer system memory. This is the first of a multi-part assignment concluding with a simple computing machine capable of executing real programs compiled with `g++`. The purpose is to gain an understanding of a machine, its instruction set and how its features are used by realistic programs written in C/C++.

1 Problem Description

To simulate a computer system's memory, create a class to represent a memory whose size is defined at run-time via command-line argument.

Your memory class will include utility member functions to load it by reading a binary file, print its contents with a hex dump, a method to determine if a given address is *legal*, and methods that allow a caller to read or write 8, 16 and 32-bit values from (or to) any legal address.

Your program will accept parameters from the command line, read data from a file and print all of its non-error output to standard out (aka stdout) via `std::cout` and usage or file loading error messages to standard error (aka stderr) via `std::cerr`. (No other output may be printed to stderr. Note that `check_illegal()` warnings are neither simulator nor user *errors* and therefore they must be written to stdout.)

2 Files You Must Write

You will write a C++ program suitable for execution on `hopper.cs.niu.edu` (or `turing.cs.niu.edu`.)

Your source files *MUST* be named exactly as shown below or they will fail to compile and you will receive zero points for this assignment.

Create a project directory for this assignment and place within it the source files defined below.

`main.cpp` Your `main()` and `usage()` function definitions will go here.

`hex.h` The declarations of your hex formatting class will go here.

`hex.cpp` The definitions of your `hex` class member functions will go here.

`memory.h` The definition of your `memory` class will go here.

`memory.cpp` The `memory` class member function definitions will go here.

2.1 `main.cpp`

You will be provided the code for a suitable `main()` function for this assignment.

Do not alter the code it as its output must match the reference key or else your output will be graded as wrong.

You must add Doxygen comments where appropriate.

The provided `usage()` function prints an appropriate "Usage" error message and "Pattern" to stderr and terminates the program in the traditional manner as discussed here:

https://en.wikipedia.org/wiki/Usage_message

usage() Function

```
1  static void usage()
2  {
3      cerr << "Usage: rv32i [-m hex-mem-size] infile" << endl;
4      cerr << "    -m specify memory size (default = 0x100)" << endl;
5      exit(1);
6  }
```

main() Function

```
1  int main(int argc, char **argv)
2  {
3      uint32_t memory_limit = 0x100;  // default memory size is 0x100
4
5      int opt;
6      while ((opt = getopt(argc, argv, "m:")) != -1)
7      {
8          switch(opt)
9          {
10         case 'm':
11             {
12                 std::istringstream iss(optarg);
13                 iss >> std::hex >> memory_limit;
14             }
15             break;
16         default:
17             usage();
18         }
19     }
20
21     if (optind >= argc)
22         usage();    // missing filename
23
24     memory mem(memory_limit);
25     mem.dump();
26
27     if (!mem.load_file(argv[optind]))
28         usage();
29
30     mem.dump();
31
32     cout << mem.get_size() << endl;
33     cout << hex::to_hex32(mem.get8(0)) << endl;
34     cout << hex::to_hex32(mem.get16(0)) << endl;
35     cout << hex::to_hex32(mem.get32(0)) << endl;
36     cout << hex::to_hex0x32(mem.get8(0)) << endl;
37     cout << hex::to_hex0x32(mem.get16(0)) << endl;
38     cout << hex::to_hex0x32(mem.get32(0)) << endl;
```

```
39      cout << hex::to_hex8(mem.get8(0)) << endl;
40      cout << hex::to_hex8(mem.get16(0)) << endl;
41      cout << hex::to_hex8(mem.get32(0)) << endl;
42      cout << hex::to_hex0x32(mem.get32(0x1000)) << endl;
43
44      mem.set8(0x10, 0x12);
45      mem.set16(0x14, 0x1234);
46      mem.set32(0x18, 0x87654321);
47
48      cout << hex::to_hex0x32(mem.get8_sx(0x0f)) << endl;
49      cout << hex::to_hex0x32(mem.get8_sx(0x7f)) << endl;
50      cout << hex::to_hex0x32(mem.get8_sx(0x80)) << endl;
51      cout << hex::to_hex0x32(mem.get8_sx(0xe3)) << endl;
52      cout << hex::to_hex0x32(mem.get16_sx(0xe0)) << endl;
53      cout << hex::to_hex0x32(mem.get32_sx(0xe0)) << endl;
54
55      mem.dump();
56      return 0;
57  }
```

2.2 `hex.h` and `hex.cpp`

These files will contain a class with some utility functions for formatting numbers as hex strings for printing.

Your `hex.h` file must contain the following class plus header guards and appropriate Doxygen comments:

hex.h

```
1  class hex
2  {
3  public:
4      static std::string to_hex8(uint8_t i);
5      static std::string to_hex32(uint32_t i);
6      static std::string to_hex0x32(uint32_t i);
7  };
```

Your `hex.cpp` file must contain the implementation of the three method functions.

An observation: This class is to be used for simplifying the application. Your code must use it to format a hex value any time it needs to do so. Therefore the `std::hex` I/O manipulator must never appear in any file other than `hex.cpp` (and in main() when reading the command-line args.)

- `std::string to_hex8(uint8_t i);`

  This function must return a `std::string` with exactly 2 hex digits representing the 8 bits of the `i` argument.

  The following code snipit is one way to format an 8-bit integer into a 2-character hex string with a leading zero:

  ```
  std::string hex::to_hex8(uint8_t i)
  {
      std::ostringstream os;
      os << std::hex << std::setfill('0') << std::setw(2) << static_cast<uint16_t>(i);
      return os.str();
  }
  ```

Note that the `static_cast` is necessary here to prevent the insertion operator (`<<`) from treating the 8-bit integer as a character and printing it incorrectly. (The printing of other integer sizes does not have this problem and therefore can be printed without such a cast.)

- `std::string to_hex32(uint32_t i);`

  This function must return a `std::string` with 8 hex digits representing the 32 bits of the `i` argument.

- `std::string to_hex0x32(uint32_t i);`

  This function must return a `std::string` beginning with `0x`, followed by the 8 hex digits representing the 32 bits of the `i` argument. It must be implemented by creating a string by concatenating a `0x` to the output of `to_hex32()` like this:

```
return std::string("0x")+to_hex32(i);
```

2.3 `memory.h` and `memory.cpp`

Your `memory.h` file must include the following class definition plus header guards and appropriate Doxygen comments.

Note that failure to implement this design accurately will cause *significant* problems with future assignments that you will implement by extending this one!

class memory

```
1  class memory : public hex
2  {
3  public:
4      memory(uint32_t s);
5      ~memory();
6
7      bool check_illegal(uint32_t addr) const;
8      uint32_t get_size() const;
9      uint8_t get8(uint32_t addr) const;
10     uint16_t get16(uint32_t addr) const;
11     uint32_t get32(uint32_t addr) const;
12
13     int32_t get8_sx(uint32_t addr) const;
14     int32_t get16_sx(uint32_t addr) const;
15     int32_t get32_sx(uint32_t addr) const;
16
17     void set8(uint32_t addr, uint8_t val);
18     void set16(uint32_t addr, uint16_t val);
19     void set32(uint32_t addr, uint32_t val);
20
21     void dump() const;
22
23     bool load_file(const std::string &fname);
24
25 private:
26     std::vector<uint8_t> mem;
27 };
```

You should feel free to inline any methods where you think it is appropriate.

- `memory(uint32_t siz);`

  Allocate `siz` bytes in the `mem` vector and initialize every byte/element to `0xa5`.

  Implement the following rounding logic (before allocating the `siz` elements) to make the job of formatting and aligning your last line of output in your `dump()` method much, *much* easier:

  ```
  siz = (siz+15)&0xfffffff0;  // round the length up, mod-16
  ```

  Note that initializing the `mem` vector can then be done immediately after rounding up `siz` by calling the `resize(count, value)` method on the `mem` vector. See: std::vector::resize.

- `~memory();`

  In the destructor clean up anything necessary.

- `bool check_illegal(uint32_t i) const;`

  Return `true` if the given address `i` does not represent an element that is present in the `mem` vector. (ie. Is there actually a byte at the given address or not?)

  If the given address is *not* within the range of valid addresses of the simulated memory then print a warning message to stdout formatted as shown below:

  ```
  WARNING: Address out of range: 0x00001000
  ```

  and return true.

  Obviously, formatting this warning message will involve using your `hex::to_hex0x32()` function.

- `uint32_t get_size() const;`

  Return the (rounded up) number of bytes in the simulated memory.

- `uint8_t get8(uint32_t addr) const;`

  Check to see if the given `addr` is in your `mem` by calling `check_illegal()`. If `addr` is in the valid range then return the value of the byte from your simulated memory at the given address. If `addr` is *not* in the valid range then return zero to the caller.

  Note that this is the *only* code that will ever read values from the `mem` vector.

- `uint16_t get16(uint32_t addr) const;`

  This function must call your `get8()` function twice to get two bytes and then combine them in little-endian[1] order to create a 16-bit return value. Because you are using your `get8()` function, the job of validating the addresses of the two bytes will be taken care of there. Do not redundantly check the validity in this function.

- `uint32_t get32(uint32_t addr) const;`

  This function must call `get16()` function twice and combine the results in little-endian order similar to the implementation of `get16()`.

[1] See RVALP and/or Wikipedia for a discussion of *little-endian* order.

- `int32_t get8_sx(uint32_t addr) const;`

  This function will call `get8()` and then return the sign-extended value of the byte as a 32-bit signed integer.

- `int32_t get16_sx(uint32_t addr) const;`

  This function will call `get16()` and then return the sign-extended value of the 16-bit value as a 32-bit signed integer.

- `int32_t get32_sx(uint32_t addr) const;`

  This function will call `get32()` and then return the value as a 32-bit signed integer.

  Hint: Do it like this: `return get32(addr);`

- `void set8(uint32_t addr, uint8_t val);`

  This function will call `check_illegal()` to verify the `addr` argument is valid. If `addr` is valid then set the byte in the simulated memory at that address to the given `val`. If `addr` is not valid then discard the data and return to the caller.

  Note that this, and the constructor, are the *only* code that will ever write values into the `mem` vector.

- `void set16(uint32_t addr, uint16_t val);`

  This function will call `set8()` twice to store the given `val` in little-endian order into the simulated memory starting at the address given in the `addr` argument.

- `void set32(uint32_t addr, uint32_t val);`

  This function will call `set16()` twice to store the given `val` in little-endian order into the simulated memory starting at the address given in the `addr` argument.

- `void dump() const;`

  Dump the entire contents of your simulated memory in hex with the corresponding ASCII[2] characters on the right *exactly, space-for-space* in the format shown in the output section below.

  In order to format the ASCII part of the dump lines, fetch a byte from the memory and then use `isprint(3)` to determine if you are to show an ASCII character or a dot (`.`) when the byte does not have a valid printable value:

  ```
  uint8_t ch = get8(i);
  ch = isprint(ch) ? ch : '.';
  ```

  This code fragment will leave the character to be printed in the ASCII portion of the dump in the `ch` variable. The `isprint()` function is a standard C library function you can read about in the on line manual or google it.

- `bool load_file(const std::string &fname);`

  Open the file named `fname` in binary mode and read its contents into your simulated memory. You may open a file in binary mode like this:

[2] See RVALP and/or Wikipedia for a discussion of the ASCII character set.

```
std::ifstream infile(fname, std::ios::in|std::ios::binary);
```

If the file can not be opened, then print a suitable message to stderr including the name of the file and return `false`:

```
Can't open file 'testdata' for reading.
```

You must make certain that the file can fit into your memory! One simple way to do that is to read the file one byte at-a-time and check the byte address before you write to it by calling `check_illegal()`. If the address is valid, keep going. If the address is not valid, then print the following message to stderr, close the file, and return `false`:

```
Program too big.
```

If the file loads OK then close the file and return `true`.

In order to read the file contents with the extraction operator (`>>`), you will want to set `noskipws` before reading from it like this:[3]

```
uint8_t i;
infile >> std::noskipws;
for (uint32_t addr = 0; infile >> i; ++addr)
{
    ...
}
```

- `std::vector<uint8_t> mem;`

  A vector of bytes representing the simulated memory. Initialize it with the given size in your constructor.

  The element at `mem[0]` represents the byte in the simulated memory at address zero. The element at `mem[1]` represents the byte in the simulated memory at address one and so on.

  The legal memory addresses are those that have elements present in the `mem` vector.

3 Input

Your program will accept an optional memory-size argument and a filename on the command line as shown in the `main()` code snipit above.

The Usage statement for this application is:

`memsim [-m hex-mem-size] filename`

The optional `-m` argument is a hex number representing the amount of memory to simulate. When not present, the default size must be `0x100`.

The last argument is the name of a file to load into the simulated memory.

Some test files may be provided for you. It is your responsibility to create your own test files as you see fit to verify the proper operation of your code.

It should go without saying that you should try loading files with sizes that are less than, exactly equal to, and greater than the number of simulated memory bytes.

[3] See https://en.cppreference.com/w/cpp/io/manip/skipws for more information.

4 Output

Your program's output will be a dump of the simulated memory after it has been constructed then again after it has been loaded. Then and some lines of output from the test calls to the `setX()`, `getX()` and `to_hexX()` functions and another dump to know that the `setX()` functions are working properly.

For example if your program is executed like this with a file that contains "hello world 1 2 3 4" on a line by itself then:

```
./memsim -m 2e hello.in
```

will display the following output:

Sample Run

```
00000000: a5 a5 a5 a5 a5 a5 a5 a5  a5 a5 a5 a5 a5 a5 a5 a5 *................*
00000010: a5 a5 a5 a5 a5 a5 a5 a5  a5 a5 a5 a5 a5 a5 a5 a5 *................*
00000020: a5 a5 a5 a5 a5 a5 a5 a5  a5 a5 a5 a5 a5 a5 a5 a5 *................*
00000000: 68 65 6c 6c 6f 20 77 6f  72 6c 64 20 31 20 32 20 *hello world 1 2 *
00000010: 33 20 34 0a a5 a5 a5 a5  a5 a5 a5 a5 a5 a5 a5 a5 *3 4.............*
00000020: a5 a5 a5 a5 a5 a5 a5 a5  a5 a5 a5 a5 a5 a5 a5 a5 *................*
48
00000068
00006568
6c6c6568
0x00000068
0x00006568
0x6c6c6568
68
68
68
WARNING: Address out of range: 0x00001000
WARNING: Address out of range: 0x00001001
WARNING: Address out of range: 0x00001002
WARNING: Address out of range: 0x00001003
0x00000000
0x00000020
WARNING: Address out of range: 0x0000007f
0x00000000
WARNING: Address out of range: 0x00000080
0x00000000
WARNING: Address out of range: 0x000000e3
0x00000000
WARNING: Address out of range: 0x000000e0
WARNING: Address out of range: 0x000000e1
0x00000000
WARNING: Address out of range: 0x000000e0
WARNING: Address out of range: 0x000000e1
WARNING: Address out of range: 0x000000e2
WARNING: Address out of range: 0x000000e3
0x00000000
00000000: 68 65 6c 6c 6f 20 77 6f  72 6c 64 20 31 20 32 20 *hello world 1 2 *
00000010: 12 20 34 0a 34 12 a5 a5  21 43 65 87 a5 a5 a5 a5 *. 4.4...!Ce.....*
00000020: a5 a5 a5 a5 a5 a5 a5 a5  a5 a5 a5 a5 a5 a5 a5 a5 *................*
```

Note that if you create a test file on Windows, then the byte whose value is `0x0a` (the newline

character) might be different than shown above.[4]

Other examples may available on the course web site.

5 How To Hand In Your Program

When you are ready to turn in your assignment, make sure that the only files in your project directory is/are the source files defined and discussed above. Then, in the parent of your project directory, use the `mailprog.463` command to send the contents of the files in your project directory to your TA. For example, if your project directory is called `memsim` then run mailprog like this:

```
mailprog.463 memsim
```

6 Grading

The grade you receive on this programming assignment will scored according to the syllabus and its ability to compile and execute on the Computer Science Department's computer.

It is your responsibility to create your own test data files suitable for testing your program thoroughly.

When we grade your assignment, we will compile it on `hopper.cs.niu.edu` using these exact commands:

Compiling Your Assignment

```
1 g++ -g -ansi -pedantic -Wall -Werror -Wextra -std=c++14 -c -o main.o main.cpp
2 g++ -g -ansi -pedantic -Wall -Werror -Wextra -std=c++14 -c -o memory.o memory.cpp
3 g++ -g -ansi -pedantic -Wall -Werror -Wextra -std=c++14 -c -o hex.o hex.cpp
4 g++ -g -ansi -pedantic -Wall -Werror -Wextra -std=c++14 -o memsim main.o memory.o hex.o
```

Your program will then be run multiple times using different memory sizes and test data files and the output compared against a reference implementation of this assignment.

[4] Text files created on DOS/Windows machines have different line endings than files created on Unix/Linux. DOS uses carriage return and line feed ("\r\n") as a line ending, while Unix uses just a line feed ("\n").