C++ Assignments

Vayavya Labs Pvt. Ltd.

Version 1.3, Feb 15 2019 / 14:17:37

1. General Information

1.1. Copyright Notice

1.2. Dos and don'ts

You are required to address the assignments described in this document yourself. External help should NOT be solicited - either from an individual, a company/organization or online forums. It is our expectation that you are able to carry out the tasks autonomously.

Referring to a language reference manual is allowed and encouraged. You can also consult external forums for pointed queries (e.g. "How to set file buffering to NONE in Python?") - but you are expected to completely understand the code that you finally submit. Blind copy-paste is not expected and defeats the purpose of this exercise. Even the code-snippets in this document must be typed out again and not copy-pasted.

The assignment implementation should be cleanly separated into header/source files. Use `.cpp` extension for C++ sources, `.h` for C++ header implementation. Other normal guidelines apply: use meaningful names, indent code correctly, provide comments. Use only spaces for indentation.

Each assignment should be done along with one or more test-case. The tests must be done in a separate file - and must (generally) run with no manual intervention. Provide a makefile for compiling sources to an executable. Sources should compile with no warnings (using `-Wall` with `g++`).

If you have any feedback or questions, please direct them to your contact person in Vayavya Labs.

2. Assignments

2.1. A simplified `std::list`

References:

- `std::list` reference: https://en.cppreference.com/w/cpp/container/list
- PIMPL idiom: https://en.cppreference.com/w/cpp/language/pimpl

Implement a class `simple_list` with the following features (refer to `std::list` documentation for details on what the methods should do):

- Is a list of `int` types
- Supports forward iteration (uses a singly linked list underneath)
- Uses `new`/`delete` internally for allocation / deallocation
- Provides a default constructor, destructor, assignment operator
- Provides `empty` and `max_size` methods
- Provides the following iterator methods: `begin`, `end`
- Provides the `front`, `push_front`, `erase_after` and `insert_after` methods
- Provide the required overloads so that `std::cout << list_obj` will print the list contents to `stdout`

The class should be implemented in `simple_list.cpp`/`simple_list.h` files. The test case should in a separate file, `test_simple_list.cpp`.

Additional challenges:

- Modify `simple_list` to use the PIMPL idiom - by moving implementation specific aspects to an other class, `simple_list_impl`.
- Modify `simple_list` to be a template class - templatized on the element type. What happens when the body of the methods are left in the `simple_list.cpp` file?

2.2. 40-bit integer type

Implement a datatype called `acc40` that behaves as a 40 bit signed integer with following features:

- Support following arithmetic operators: binary `+`, binary `-`, unary `-`, `*`, `/`
- Support following logical operators: `&&`, `||`
- Support relational operators: `!=`, `==`, `<`, `>`, `<=`, `>=`

- Support assignment operator
- Have a default constructor (sets value to 0) and copy constructor
- Conversion from/to integer types must be supported
- Provide the required overloads so that `std::cout << acc40_obj` will print the value to `stdout`

The class should be implemented in `acc40.cpp`/`acc40.h` files. The test case should in a separate file, `test_acc40.cpp`.

Additional challenges:

- How will you modify the class to support overflow detection? i.e., when result of an arithmetic operation is more than 40 bits, a flag must be set (can be obtained using, `is_overflow` method on the object).
- If the task were to create a general template class, `template class acc<int N>` for N-bit arithmetic, how will the `acc40` implementation change? Would you consider any template specializations for optimizations here? Consider cases like `acc<1>` vs `acc<128>`. Actual implementation not required - providing a design approach would be sufficient.

2.3. Expression Trees

References:

- Binary Expression Trees:
  - https://en.wikipedia.org/wiki/Binary_expression_tree
  - https://web.archive.org/web/20170119094603/http://www.brpreiss.com/books/opus5/html/page264.html
- Visitor Pattern:
  - https://cpppatterns.com/patterns/visitor.html
  - https://gieseanw.wordpress.com/2018/12/29/stop-reimplementing-the-virtual-table-and-start-using-double-dispatch/

Consider an arithmetic binary expression with:

- Literal numbers, e.g.: `100`, `-42`)
- Arithmetic operators: binary `+`, binary `-`, unary `-`, `*`, `/`
- Variable references (following C-syntax), e.g.: `foo`, `bar42`

Implement an expression tree to store a representation of such an arithmetic expression. Some code-snippets are provided below as a start - modify/extend them as needed for the exercise. Implement constructors (default, copy), destructor if required. Note that when a Tree object gets deleted, it should free all the sub-nodes as well.

```
// Base class for all nodes
class Node
{
   // ...
};

class Num : public Node
{
   public:
      LiteralNumber(int n);
};

class Var : public Node
{
   // ...
};

class OpPlus : public Node
{
   // ...
};

// etc...

class Tree
{
   public:
      Tree(Node * current, Tree *left = NULL, Tree *right = NULL);
      Tree * left();  // Return left branch, NULL if empty
      Tree * right(); // Return right branch, NULL if empty
                      // Note: right branch for unary operators is NULL
      Node * node();  // Return current node reference
   private:
      // ...
};
```

An expression such as 1+2*3 can be represented as a tree as follows:

```
Tree t(new OpPlus
      , new Tree(new Num(1))
      , new Tree(new OpMultiply
         , new Tree(new Num(2))
         , new Tree(new Num(3))));
```

Overload the `operator<<` on `ostream` so a `Tree` object can be printed. How will you design the classes to let each class derived from `Node` to implement the "print" for that node?

```
std::cout << t << std::endl; // Should print: (1+(2*3))
```

Add a constructor to `Tree` class that constructs the tree from an prefix notation:

```
// Declaration:
   Tree::Tree(std::vector<Node *> prefix_expr);

// Usage:
   std::vector<Node *> expr = {new OpPlus
      , new Num(1)
      , new OpMultiply
      , new Num(2)
      , new Num(3)};
   Tree t(expr);
```

Add an `evaluate` method to the `Tree` class that will simplify the expression by evaluating all sub-trees formed only by operations on literal numerics.

```
// Declaration:
   Tree::evaluate();

// Usage:
   // Before, t = a + 1 + 2 + b*(2*3) - a
   t.evaluate();
   // After, t = a + 3 + b*6 - a
   // Note that the "a" term is not cancelled out
   // Objective is to only evaluate sub-trees formed using literal numerics
```

2.4. Technical Article

Write a technical article (in style of a blog). You can pick your own topics, giving a few below as suggestions. The article/blog should provide a good coverage on the topic. It can include code snippets, diagrams, general gotchas/pitfalls, user guidelines, etc. It should be original content - not copy - paste.

Topic suggestions:

- `const` in C++
- Use of `std::bind` (with an overview of function pointers - including pointers to member functions)
- Introduction to design patterns (pick some - visitor, factory, model-view-controller, etc)
- Introduction to smart pointers in C++
- Introduction to C++11 move semantics - rvalue and universal references, `std::move`, move constructors
- Introduction to C++11 features - Automatic type deduction using "auto" keyword, Enhancements to enum, Initializer lists, Explicitly defaulted and deleted special member functions and override specifier, Lambdas

You are strongly encouraged to write the article in AsciiDoctor (https://asciidoctor.org/) or Markdown (https://daringfireball.net/projects/markdown/) syntax - but if that is very distracting, you can use Google Docs.

2.5. Buttons on a Frame

Say you have a `class Button;` that shows a GUI button on screen. The `Button` class lets you register a function that will be called when the user clicks the button.

```
// Contents of button.h
class Button
{
   public:
      typedef void (*fptr_t)(void);
      // Take function pointer to call when user clicks on button
      void on_click(fptr_t f);
};
```

You should write a class that, when instantiated, will create N such buttons (where N is a constructor time parameter).

```
// Contents of MyFrame.h
class MyFrame
{
   public:
      MyFrame(int N) : m_buttons(N), m_total_clicks(0), m_clicks(N)
      {
         for(int i=0; i<N; i++) {
            m_clicks[i] = 0;
         }

         // ...
      }
   public:
      std::vector<Button> m_buttons;
      int m_total_clicks;
      std::vector<int> m_clicks;
};
```

The requirement is the following: Each time the user clicks the `i'th` button, the `m_clicks[i]` and `m_total_clicks` should both get incremented and the updated values need to be printed on the screen. It is acceptable to modify `Button` class if required - but it should remain generic.

2.6. LC3 and Memory

References:

- LC3 ISA presentation: http://www.cs.utexas.edu/users/fussell/cs310h/lectures/Lecture_10-310h.pdf
- The LC-3 ISA reference: http://highered.mheducation.com/sites/dl/free/0072467509/104691/pat67509_appa.pdf
- LC3 Examples: http://people.cs.georgetown.edu/~squier/Teaching/HardwareFundamentals/LC3-trunk/docs/LC3-AssemblyManualAndExamples.pdf
- Wikipedia article: https://en.wikipedia.org/wiki/LC-3
- LC3 Assembler: https://github.com/davedennis/LC3-Assembler

2.6.1. LC3 Introduction

The LC-3 specifies a word size of 16 bits for its registers and uses a 16-bit addressable memory with maximum addressable memory of 2^{16} locations. The register file contains eight registers, referred to as `R0` through `R7`. All of the registers are general-purpose in that they may be freely used by any of the instructions that can write to the register file. There are other registers that influence the operation: 16-bit Program Counter (`PC`) and 16-bit Processor Status Register (`PSR`) - which includes 3 x 1-bit condition codes and other information.

Instructions are 16 bits wide and have 4-bit opcodes. The instruction set defines instructions for fifteen of the sixteen possible opcodes, though some instructions have more than one mode of operation. The architecture is a load-store architecture; values in memory must be brought into the register file before they can be operated upon.

All data in the LC-3 is assumed to be stored in a two's complement representation; there is no separate support for unsigned arithmetic. The LC-3 has no native support for floating-point numbers.

Arithmetic instructions available include addition (`ADD`), bitwise `AND`, and bitwise `NOT`, with the first two of these able to use both registers and sign-extended immediate values as operands. These operations are sufficient to implement a number of basic arithmetic operations, including subtraction (by negating values) and bitwise left shift (by using the addition instruction to multiply values by two). The LC-3 can also implement any bitwise logical function, because `NOT` and `AND` together are logically complete.

Memory accesses can be performed by computing addresses based on the current value of the program counter (`PC`) or a register in the register file; additionally, the LC-3 provides indirect loads and stores, which use a piece of data in memory as an address to load data from or store data to. Values in memory must be brought into the register file before they can be used as part of an arithmetic or logical operation.

The LC-3 provides both conditional and unconditional control flow instructions. Conditional branches are based on the arithmetic sign (negative, zero, or positive) of the last piece of data written into the register file. Unconditional branches may move execution to a location given by a register value or a `PC`-relative offset. Three instructions (`JSR`, `JSRR`, and `TRAP`) support the notion of subroutine calls by storing the address of the code calling the subroutine into a register before changing the value of the program counter. The LC-3 does not support the direct arithmetic comparison of two values. Computing the difference of two register values requires finding the negated equivalence of one register value and then, adding the negated number to the positive value in the second register. The difference of the two registers would be stored in one of the 8 registers available for the user.

2.6.2. Modeling Requirements

In this assignment, you will implement a ISS (Instruction Set Simulator) of LC-3.

The memory will be separated from the ISS. The ISS will implement Harvard architecture, with separate access paths for instruction and data. A real process will have an interface like AMBA AXI or AHB. In this case, the ISS will have a abstract interface as described below.

The class bus_if abstracts the bus interface and allows single element reads and writes.

```
// bus_if.h
class bus_if
{
   public:
      enum {BUS_OK, BUS_ERROR} status_t;
      virtual status_t read(uint16 addr, uint16 &data) = 0;
      virtual status_t write(uint16 addr, uint16 data) = 0;
};
```

The LC3 class skeleton should be as follows:

```
// Files: LC3.h and LC3.cpp
class LC3
{
   public:
      bus_if *iport; // Instruction access, only reads
      bus_if *dport; // Data access for load/store, reads and writes

      // Resets the LC3: R0-R7, PC gets set to 0, flags cleared
      void reset();

      // Crunches the ISS forward by one "step": instruction fetch,
      // decode and execute
      void step();
   private:
      std::vector<uint16> m_R;
      uint16 m_PC;

      // Various fields of PSR register
      bool m_privilege;
      int  m_priority;
      bool m_N;
      bool m_Z;
      bool m_P;
};
```

As indicated, the model will use iport for instruction access and dport for load/stores. The step function will step the state forward by one instruction fetch/decode/execute cycle.

```
void LC3::step()
{
   // ...
   iport->read(m_PC, instruction);
   decode_and_execute(instruction);
}
```

The LC3 will need a memory model to execute - this will be implemented as a separate class. The memory model implements the bus interface. The skeleton for the memory model is as shown below:

```
// Files: Memory.h and Memory.cpp
class Memory : public bus_if
{
   public:
      virtual status_t read(uint16 addr, uint16 &data);
      virtual status_t write(uint16 addr, uint16 data);

      // Load the memory with values as specified in file
      // Return "true" on success
      bool load(std::string file_name);
   private:
      // ...
};
```

The `load` method is to initialize the memory contents. The format of the file given as argument to the method is as follows:

- Each line of the file is either a comment, or an address, or memory content or a blank line
- Comments start with `#` in the first column of the row, addresses with an `A` and memory data with `D`.
- Spaces are not allowed, except as part of comment line.
- Comments are ignored from being processed further
- The data following an address line is stored at that address
- There can be several lines of data after an address line - these are placed contiguously

An example file is shown below:

```
# This is a comment line
# Blanks like below are ok

# Line below specifies the address
A 0x3000

# Data follows. 0x0000 is stored at 0x3000,
# 0x1111 at 0x3001, ..., 0x9999 at 0x3009
D 0x0000
D 0x1111
D 0x2222
D 0x3333
D 0x4444
D 0x5555
D 0x6666
D 0x7777
D 0x8888
D 0x9999

# Line below specifies a new address
A 0x1000

# Data follows. 0x0000 is stored at 0x1000,
# 0x1111 at 0x1001
D 0x0000
D 0x1111
```

If there are any errors during load, the method prints the error and returns `false`. Execution should stop upon load failure.

Finally, the `main` function should be written as follows.

```
// File: main.cpp
// The executable should be run with the file to be loaded in memory as
// argument, like:
//   $ lc3sim memfile
int main(int argc, char *argv[])
{
   // TODO: Check if argc is right, etc..

   // Instantiate components
   LC3 lc3;
   Memory memory;

   // "Connect" them
   lc3.iport = &memory;
   lc3.dport = &memory;

   // Initialize
   memory.load(argv[1]);

   // Simulate
   while(1)
   {
      lc3.step();
      // TODO: Add debug prints to monitor progress
   }
}
```