CS 352 Homework
Baseball

1 Overview

Your task in this homework is learn about number representation and conversion between between bases. You will do this in the context of writing a calculator named `Baseball` that accepts terms in bases 2 through 36 and a handful of operations. Such a calculator belongs in every programmer's toolbelt. Values may be captured in variables and referenced in subsequent expressions. Consider the following execution to see how `Baseball` operates:

```
> 6 -> 2
110
> 777_8
511
> 11 << 2
44
> 99_16
153
> :flags <- 1101_2
13
> :mask <- 110_2 -> 2
110
> :flags & :mask
100
> 253 -> 16
fd
> ~101_2 -> 2
11111111111111111111111111111010
```

2 Stringstream

You will occasionally wish to concatenate integers and strings or parse numbers from strings as you complete this assignment. Java makes this pretty simple with the concatenation operator and `Integer.parseInt` or `Scanner`. In C++, we can use the `stringstream` class, which effectively lets us write to and read from a `string`.

```
#include <sstream>
using std::stringstream;
...

// To concatenate
stringstream out;
out << 404 << ".html";
string concatenated = out.str(); // stores "404.html"

// To parse
stringstream in("560");
unsigned int n;
in >> n; // stores 560
```

3 References

Pointers in C are great for sharing data cheaply between functions. However, they wear their pointer badges proudly, and they must be dereferenced with the `*` or `->` operators to get at their underlying value. In C++, we favor *references* over pointers. These provide the cheap sharing of pointers without pointer syntax. The only syntactic element they need is the `&` annotating the reference's type declaration. Consider the example of `triple`, which reads from and writes to its parameter, which is passed by reference:

```
void triple(int &n) {
  n = n + n + n;
}

int main(int argc, char **argv) {
  int x = 10;
  triple(x);
  std::cout << x << std::endl; // prints 30
  return 0;
}
```

Almost all the methods you complete in this homework will use reference parameters. When the parameter is read-only, it must be marked with the `const` qualifier.

4 Requirements

Implement the files, classes, or routines described below. Place all files in a directory named `baseball` at the top-level of your repository.

4.1 Makefile

Create a `Makefile` to compile your code. Note the capital M. Its default (topmost) rule should build an executable named `main` that runs your `main` function. To be most useful, this target should depend on `Baseball.o`, `main.o`, and `Makefile`. If any of these dependencies are newer than `main`, it rebuilds the executable.

Provide two general rules for compiling `*.o` files from their source, one that includes a header and one that does not:

```
%.o: %.cpp Makefile
  $(CPP) $(CFLAGS) -c -o $@ $<

%.o: %.cpp %.h Makefile
  $(CPP) $(CFLAGS) -c -o $@ $<
```

`$@` is a builtin variable that refers to the rule's target (the `*.o` file). `$<` refers to the leftmost dependency (the `*.cpp` file). Define `CPP` to be `g++` and `CFLAGS` to include debugging information (`-g`) and invoke a modern C++ standard (`-std=c++11`).

Also provide a `clean` rule like the following to dispose of any derived files:

```
rm -f *.o main
```

4.2 Main

Write a C++ file `main.cpp` with a `main` function, which you are encouraged to use to test your code. In fact, we will even provide a read-eval-print loop (REPL) that you may freely use.

```
#include <iostream>
#include "Baseball.h"

int main(int argc, char **argv) {
  Baseball calc;

  std::cout << "> ";
  std::string line;
  while (std::getline(std::cin, line)) {
    try {
      std::cout << calc.Evaluate(line) << std::endl;
    } catch (const char *s) {
      std::cerr << s << std::endl;
    } catch (const std::string &s) {
      std::cerr << s << std::endl;
    }
    std::cout << "> ";
  }
  std::cout << std::endl;

  return 0;
}
```

4.3 Baseball

Write class `Baseball` to parse and evaluate calculator expressions. It has the following:

1. No `main` function. Since C++ allows only one `main` function to exist within a scope and since the SpecChecker defines one to test `Baseball`, you cannot define one except in `main.cpp`.
2. Its declaration in file `Baseball.h` and its method definitions in file `Baseball.cpp`.
3. Method `EvaluateDigit`, which is `static` and which accepts one parameter: a digit of type `char`. It returns the numeric value of the given digit, presumably a decimal digit or a letter. If the digit is one of the 10 decimal digits, return its numeric value in $[0, 9]$. (Note that a decimal digit character interpreted numerically has a value in $[48, 57]$, as originally defined in ASCII. To compute a decimal digit's real decimal value, subtract off `'0'`.) If the digit is a letter of either case, make it such that `'A'` returns 10 and `'Z'` returns 35, with the intervening letters returning the intervening numbers. For example:
   - `Baseball::EvaluateDigit('7')` $\rightarrow 7$
   - `Baseball::EvaluateDigit('z')` $\rightarrow 35$
   - `Baseball::EvaluateDigit('C')` $\rightarrow 12$

   The result for non-digit, non-letter values is undefined, but throwing a C-string will work best with the provided `main`. For example:

```
stringstream out;
out << "Illegal digit: " << digit;
throw out.str();
```

4. Method `ValueToDigit`, which is `static` and which accepts one parameter: a value of type `unsigned char`. It returns the single digit `char` representation of the given value. If the value is below 10, it returns the corresponding decimal character. If the value is in $[10, 35)$, it returns the corresponding lowercase letter. For example:

   - `Baseball::ValueToDigit(3)` $\rightarrow$ `'3'`
   - `Baseball::ValueToDigit(13)` $\rightarrow$ `'d'`
   - `Baseball::ValueToDigit(20)` $\rightarrow$ `'k'`

   Throw a C-string for values outside the legal range.

5. Method `FromBase`, which is `static` and which accepts two parameters in this order:

   (a) a `const` reference to a token of type `string`

   (b) a base in [2, 36], of type `unsigned int`

   It returns the value of the given token as an `unsigned int`. The token is in the given base. For example, `Baseball::FromBase("ff", 16)` $\rightarrow$ 255.

   Consider the number 392 in decimal. This can be decomposed into an accumulation of successive powers of 10:

$$\begin{aligned} 392 &= 300 + 90 + 2 \\ &= 100 \times 3 + 10 \times 9 + 1 \times 2 \\ &= 10^2 \times 3 + 10^1 \times 9 + 10^0 \times 2 \end{aligned}$$

   Consider the number 75af in hexadecimal. This can be decomposed into an accumulation of successive powers of 16:

$$\begin{aligned} \text{75af}_{16} &= 16^3 \times 7 + 16^2 \times 5 + 16^1 \times \text{a} + 16^0 \times \text{f} \\ &= 16^3 \times 7 + 16^2 \times 5 + 16^1 \times 10 + 16^0 \times 15 \end{aligned}$$

   We can generalize these breakdowns into the following algorithm, which converts a number encoded as a `string` in a given base to its numeric value:

```
magnitude = 1
for each digit, from least significant to most
  value = value + magnitude * digit's worth
  magnitude = magnitude * base
```

   Results are undefined if a digit is not legal in the specified base, but we recommend you throw a C-string explaining the error.

6. Method `ToBase`, which is `static` and which accepts two parameters in this order:

   (a) a value of type `unsigned int`

   (b) a base in [2, 36], of type `unsigned int`

   It returns the representation of the given value in the given base. The return value must be a `string` because we may need to use digits beyond 0 through 9. This method is the inverse of `FromBase`. Use lowercase letters for digits greater or equal to 10. For example, `Baseball::ToBase(255, 16)` → `"ff"`.

   The representation can be computed by determining one digit at a time, from right to left, using the following algorithm:

   ```
   token = ""
   if n is 0
     token = "0"
   else
     while n > 0
       find symbol for least significant digit
       prepend symbol to token
       remove least significant digit from n
   ```

   Recall that digits can be extracted from a number using integer division and the remainder operator, both with an appropriate second operand. For example, `182 % 10` → 2, and `182 / 10` → 18.

7. Method `EvaluateLiteral`, which is `static` and which accepts one parameter: a `const` reference to a literal token of type `string`. It returns the literal's value as an `unsigned int`. The literal may appear in two forms: as a number that's implicitly in base 10 (like `"755"`) or as a number with an explicit base appearing after an underscore (like `"10110_2"` or `"773_8"`). Use an earlier method to do most of the conversion work. You will probably need to convert a `string` into an `unsigned int`.

8. Method `EvaluateVariable`, which is `const` and which accepts one parameter: a variable identifier, a reference to a `const string`. It returns the `unsigned int` value bound to the identifier from some earlier assignment. (You'll need some mechanism for associating `string`s with `unsigned int`s. We recommend `std::map`, which uses a red-black tree to organize the data for fast lookup. A hashtable API was only recently introduced to the C++ standard library.)

   Note that legal variable identifiers start with a colon character (`:`). The results are undefined if an illegal or unbounded identifier is given, but we recommend you throw an explanatory C-string.

9. Method `EvaluateTerm`, which is `const` and which accepts one parameter: a `const` reference to a token of `string`. It returns the term's value as an `unsigned int`. The token is either a variable identifier or a literal. However, the first character of the token may optionally be `~`, which indicates that the value should be negated. (C++ supports this operator too.) For example:

- `calc.EvaluateTerm("542")` → 542
- `calc.EvaluateTerm(":pi")` → 3
- `calc.EvaluateTerm("~1")` → 4294967294

The results are undefined if an illegal token is given.

10. Method `EvaluateOperation`, which is `const` and which accepts three parameters in this order:

    (a) a `const` reference to a left term of type `string`

    (b) a `const` reference to an operator symbol of type `string`, assumed to be one of `+`, `-`, `&` (bitwise and), `|` (bitwise or), `^` (exclusive bitwise or), `<<` (left shift), `>>` (right shift), `*`, `/` (integer division), or `%`.

    (c) a `const` reference to a right term of type `string`

    It evaluates the two terms and combines them according to the given operation, returning the result as an `unsigned int`. For example:

    - `calc.EvaluateOperation("110_2", "<<", "2")` → 24
    - `calc.EvaluateOperation(":pi", "+", ":pi")` → 6
    - `calc.EvaluateOperation("~100", "/", "1000000")` → 4294

    The results are undefined if the operands are illegal or the operation is unknown, but we recommend you throw a C-string.

11. Method `Tokenize`, which is `static` and which accepts one parameter: a `const` reference to an expression of type `string`. The tokens of the expression are separated by whitespace. This method returns the tokens as a `vector` of `string`s. (The counterpart in C++ of Java's `ArrayList` is `vector`.)

    In Java, you could parse the `string` on whitespace using `Scanner`. In C++, you can use `stringstream`. For example, this code prints all tokens:

```
stringstream in("it is fun to be in the same decade with you");
string token;
while (in >> token) {
  std::cout << token << std::endl;
}
```

12. Method `ExtractTargetBase`, which is `const` and which accepts one parameter: a reference to a list of an expression's tokens as a `vector` of `string`s. It returns as an `unsigned int` the target base for the expression's printed result.

    If the second-to-last token is `"->"`, then the last token is assumed to be a term specifying the target base. (*Term* is defined to be whatever `EvaluateTerm` can evaluate.) The arrow and base term are removed from the `vector`. Otherwise, the target base is implicitly 10.

13. Method `ExtractTargetID`, which is `static` and which accepts one parameter: a reference to a list of an expression's tokens as a `vector` of `string`s. It returns as a `string` the identifier of the variable to which the expression's value should be assigned.

If token 1 is `"<-"`, then token 0 is the identifier, which is assumed to start with a colon. The arrow and identifier are removed from the `vector`. Otherwise, the identifier is implicitly `":last"`, a special builtin variable meant to hold the result of the last executed expression.

14. Method `Evaluate`, which accepts one parameter: a `const` reference to a list of an expression's tokens as a `vector` of `strings`. It returns as an `unsigned int` the value of the expression.

    The tokens list is assumed to contain either a) a single term or b) a left term, an operand, and a right term. This method should defer most of its work to methods you've written previously.

15. Method `Evaluate`, which accepts one parameter: a `const` reference to an expression of type `string`. It returns as a `string` the value of the expression in the target base, storing the result in the target variable identifier. For example:

    (a) `calc.Evaluate(":pi <- 3")` stores 3 in variable `:pi` and evaluates to `"3"`.

    (b) `calc.Evaluate("8 + 2 -> 2")` stores 10 in variable `:last` and evaluates to `"1010"`.

    To contain the scope of this assignment, the expression will contain at most one two-operand operation. The expression is assumed to be legal, with each of its tokens separated by at least one whitespace character. This method should defer most of its work to methods you've written previously.

5 Later-week Submission

To qualify for later-week submission, you must provide `Makefile` and `main.cpp` and, for class `Baseball`, define the constructor, `EvaluateDigit`, `ValueToDigit`, `FromBase`, `ToBase`, and `EvaluateLiteral`.

6 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.
- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you qualified for later week submission and are resubmitting after the original deadline.