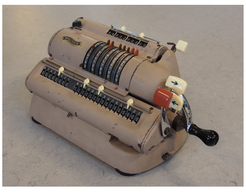
CS253 HW2: Command-Line Arguments!



Description

For this assignment, you will write a calculator program that that takes command-line arguments, performs simple arithmetic, and produces the result in the base desired.

Arguments

The first, optional argument is one of **`-h`**, **`-o`**, or **`-d`**, signifying an output base of hexadecimal, octal, or decimal. The default is decimal.

After that is a mandatory operator, one of:
- **`+`**: addition
- **`-`**: subtraction
- **`*`**: multiplication (remember to quote it)
- **`/`**: integer division
- **`@`**: distance from 45 (see below)

After that is a mandatory sequence of at least one integer, of the form:
- **`0x`***digits*: a hexadecimal integer, containing digits **`0123456789abcdef`**.
- **`0b`***digits*: a binary integer, containing digits **`01`**.
- **`0`***digits*: an octal integer, containing digits **`01234567`**.
- *digits*: (not matching any of the above patterns) a decimal integer, containing digits **`0123456789`**.

No more arguments should follow.

Operation

The operation is applied to *all* the numbers, left-to-right. For example, **`./hw2 + 10 20 30`** will calculate the result of 10+20+30, or 60. **`./hw2 / 1000 5 2 25`** will calculate the result of 1000/5/2/25, or 4. The result is displayed in the base indicated by the optional first argument.

Quoting

Unfortunately, the `*` character is special to `bash`, and will be replaced by a list of all the files in your current directory. To avoid this, type `\*` or `'*'` instead.

Distance from 45

The `@` operator returns one of its operands, namely, the operand that is furthest from 45. If the operands are equally distant from 45, the smaller number is returned.

Sample Runs

Here are sample runs, where `%` is my prompt.

```
% ./hw2 + 2 2
4
% ./hw2 -d + 1 2 3 4 5
15
% ./hw2 -d - 1 2 3 4 5
-13
% ./hw2 -h - 1 17
-10
% ./hw2 -h - 100 0b101 0x19 017
37
% ./hw2 -o / 13 007
1
% ./hw2 -o / 7 13
0
% ./hw2 '*' 2 123 1
246
% ./hw2 @ 30 70 29
70
% ./hw2 - 0b1001
9
% ./hw2 -h + 64
40
% ./hw2 -h / 64206
face
```

Output in other bases:

Observe:

```
cout        << 20 << ' ' << 30 << '\n';
cout << hex << 20 << ' ' << 30 << '\n';
cout << oct << 20 << ' ' << 30 << '\n';
cout << dec << 20 << ' ' << 30 << '\n';
```

```
20 30
14 1e
24 36
20 30
```

Requirements

- Perform truncation in integer division using the `/` operator.
- Use `int` for your calculations. The results of overflow, e.g., by multiplying or adding very large numbers, is undefined.
- The result of entering an integer that's too large for an `int`, e.g., **`0x123456789`**, is undefined.
- If you type a `*` as an argument without quoting it, that's your problem.
- Error messages:
  - go to standard error.
  - include the program name, no matter how it was compiled.
  - include the offending argument
- Produce an error message and stop the program if:
  - not enough arguments are given
  - incorrect arguments are given
  - division by zero is attempted.

- The output must end with a newline.
  - Newlines do not separate lines—newlines *terminate* lines.
- Creativity is a wonderful thing, but your output format is *not* the place for it. Your non-error output should look ***exactly*** like the output shown above. You have more leeway in error cases.
  - UPPERCASE/lowercase matters.
  - Spaces matter.
  - Blank lines matter.
  - Extra output matters.

- You may not use any external programs via `system()`, `fork()`, `popen()`, `execl()`, `execvp()`, etc.
- You may not use C-style I/O facilities, such as `printf()`, `scanf()`, `fopen()`, and `getchar()`.
  - Instead, use C++ facilities such as `cout`, `cerr`, and `ifstream`.
- You may not use dynamic memory via `new`, `delete`, `malloc()`, `calloc()`, `realloc()`, `free()`, `strdup()`, etc.
  - It's ok to *implicitly* use dynamic memory via containers such as `string` or `vector`.
- You may not use the `istream::eof()` method.
- No global variables.
  - Except for an optional single global string containing `argv[0]`.
- For readability, don't use ASCII `int` constants (`65`) instead of `char` constants (`'A'`) for printable characters.

- We will compile your program like this: **`cmake . && make`**
  - If that generates warnings, you will lose a point.
  - If that generates errors, you will lose *all* points.
- There is no automated testing/pre-grading/re-grading.
  - Test your code yourself. It's your job.
  - Even if you only change it a little bit.
  - Even if all you do is add a comment.

If you have any questions about the requirements, **ask**. In the real world, your programming tasks will almost always be vague and incompletely specified. Same here.

Tar file

- For each assignment this semester, you will create a tar file, and turn it in.
- The tar file for this assignment must be called: **`hw2.tar`**
- It must contain:
  - source files (**`*.cc`**)
  - header files (**`*.h`**) (if any)
  - **`CMakeLists.txt`**
- This command must produce the program **`hw2`** (note the dot):
  ```
  cmake . && make
  ```
- *At least* **`-Wall`** must be used every time **`g++`** runs.

How to submit your homework:

Use web checkin, or Linux checkin:

```
~cs253/bin/checkin HW2 hw2.tar
```

How to receive *negative* points:

Turn in someone else's work.