Lec 3: Integer Arithmetic

CS220: Programming Principles

Sang Kil Cha
In-Class Activity
In-Class Activity #02 (Preparation)

1. Go to the directory you cloned for Activity #01.
   - cd 2020s-activities
   - In case you have deleted the directory, just clone it again from https://github.com/KAIST-CS220/2020s-activities
2. Run git pull
3. cd 02
4. dotnet test
In-Class Activity #02 (Task)

Modify the `myfunc` function. The function takes in four integers as input, and returns the largest number from them.
Integer Types
Various Primitive Types

int64, uint64, int32, uint32, int16, uint16, uint8, int8.

But, no int128!
### Suffix for Number Literals

To represent a number literal, we use suffix to represent its type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Suffix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code> / <code>int32</code></td>
<td>no suffix / l</td>
<td><code>1</code> / <code>1l</code></td>
</tr>
<tr>
<td><code>uint32</code></td>
<td><code>u</code></td>
<td><code>1u</code></td>
</tr>
<tr>
<td><code>int64</code></td>
<td><code>L</code></td>
<td><code>1L</code></td>
</tr>
<tr>
<td><code>uint64</code></td>
<td><code>UL</code></td>
<td><code>1UL</code></td>
</tr>
<tr>
<td><code>int16</code></td>
<td><code>s</code></td>
<td><code>1s</code></td>
</tr>
<tr>
<td><code>uint16</code></td>
<td><code>us</code></td>
<td><code>1us</code></td>
</tr>
<tr>
<td><code>int8</code> / <code>sbyte</code></td>
<td><code>y</code></td>
<td><code>1y</code></td>
</tr>
<tr>
<td><code>uint8</code> / <code>byte</code></td>
<td><code>y</code></td>
<td><code>1uy</code></td>
</tr>
</tbody>
</table>
Signed vs. Unsigned

Unsigned integers cannot represent “negative numbers”.

1u - 42u = ?
Integer Overflow
Unit of Computation

In a modern desktop machine, we use a 64-bit CPU, which means that the basic computation unit of it is 64 bits (8 bytes). Most instructions in our CPU can handle only up to 64-bit numbers.
What's the Implication?

$$18446744073709551615 + 1 = 0$$
What’s the Implication?

18446744073709551615 + 1 = 0

Why? $2^{64} - 1 = 18446744073709551615$. 
Overflow By Examples

18446744073709551615UL + 1UL // 0UL
2147483647 + 1 // -2147483648
0u - 1u // 4294967295u
Detecting Overflows?

We can always detect overflows by subdividing normal and abnormal cases.

```plaintext
let z = x - y // example: subtract operation
if z <= x then // normal
else // abnormal (overflow)
```
Representing Big Numbers?

We can “emulate” big numbers with 64-bit/32-bit operations using data abstraction!¹

\[
\text{type } \text{ UInt128 } = \text{ uint64 } * \text{ uint64}
\]

¹You will learn this later.
Big vs. Normal Integers

Big integer (arbitrary-precision) arithmetic is slower, and uses more memory. Therefore, we prefer to use normal integers in most cases.

Why? You need to understand the distinction between register and memory. Take CS230 (System Programming), etc.
Big Numbers

There is a built-in data type `bigint` for representing big numbers. We use ‘I’ as a suffix for big numbers.

```
1I + 2I // 3I
1I - 2I // -1I
999999999999999999999999999999999999999999999999999I // valid
bigint 42 // bigint is a conversion function
bigint 42L = 42I // true
```
In-Class Activity #3

Modify the function `myfunc`. The function should take in two 64-bit integer (int64) values `a` and `b`, and returns `a − b`.

- In normal cases, it should return the resulting number.
- In error cases, i.e., overflow cases, it should simply return 0.
Conclusion
Further Readings

• https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html
Question?