`unsigned size_t`

was a mistake

by Andrea “6502” Griffini

`unsigned size_t` was a mistake

`unsigned`

≠

non-negative

`unsigned size_t` was a mistake

The difference of two non-negative numbers always gives a non-negative result

`unsigned size_t` was a mistake

The difference of two non-negative numbers always gives a non-negative result

NO!! … for example 3 – 4 = -1

`unsigned size_t` was a mistake

Adding a possibly negative number to a non-negative number always gives a non-negative result

`unsigned size_t` was a mistake

Adding a possibly negative number to a non-negative number always gives a non-negative result

NO!! … for example -4 + 3 = -1

`unsigned size_t` was a mistake

A non-negative number can be smaller than -1

`unsigned size_t` was a mistake

A non-negative number can be smaller than -1

No non-negative number can be smaller than -1. That is basically the **DEFINITION** of non-negative.

HOWEVER...

`unsigned size_t` was a mistake

The difference of two **`unsigned`** numbers always gives an **`unsigned`** result

`unsigned size_t` was a mistake

The difference of two **`unsigned`** numbers always gives an **`unsigned`** result

Yes!! … for example (32 bit)

$3 - 4 = 4,294,967,295 = 2^{32}-1$

`unsigned size_t` was a mistake

Adding a possibly negative number to an **`unsigned`** number always gives an **`unsigned`** result

Adding a possibly negative number to an **`unsigned`** number always gives an **`unsigned`** result

YES!! … for example (32 bit)

$-4 + 3 = 4{,}294{,}967{,}295 = 2^{32}-1$

`unsigned size_t` was a mistake

A `unsigned` can be smaller than -1

`unsigned size_t` was a mistake

A `unsigned` can be smaller than -1

YES!! … for example 4 < -1

Actually **MOST** of them are

`unsigned size_t` was a mistake

But but but…
it's because of “overflow”

`unsigned size_t` was a mistake

But but but…
it's because of “overflow”

NO!

`unsigned size_t` was a mistake

Overflow is *undefined behavior*, and happens on values too large to be represented correctly by the platform, values that correct programs don't use.

The “strange” behavior of **`unsigned`** is instead very *well defined* and happens around zero, probably the most common value used in programming.

`unsigned size_t` was a mistake

`unsigned`

≠

non-negative

`unsigned size_t` was a mistake

`unsigned`

=

modulo integer

`unsigned size_t` was a mistake

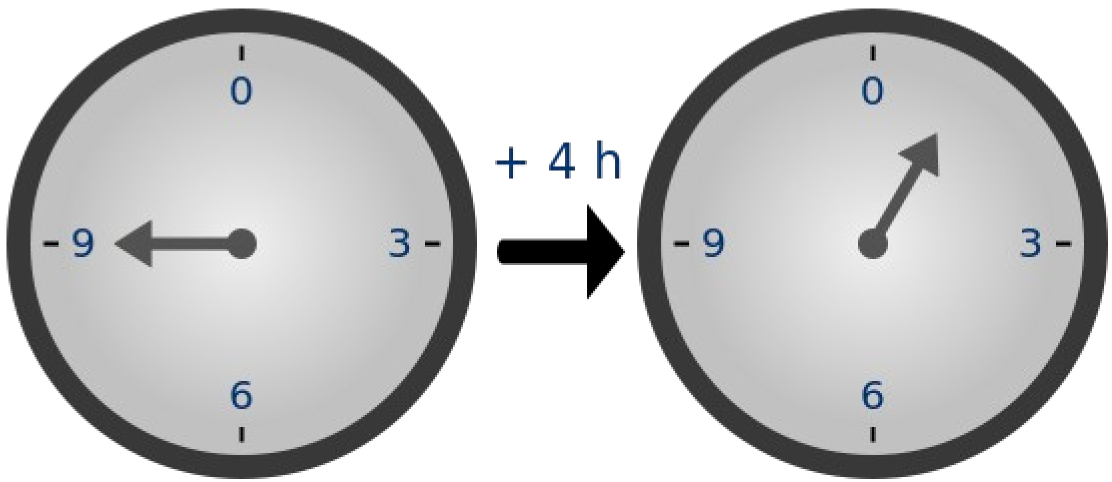
`unsigned`

=

modulo integer

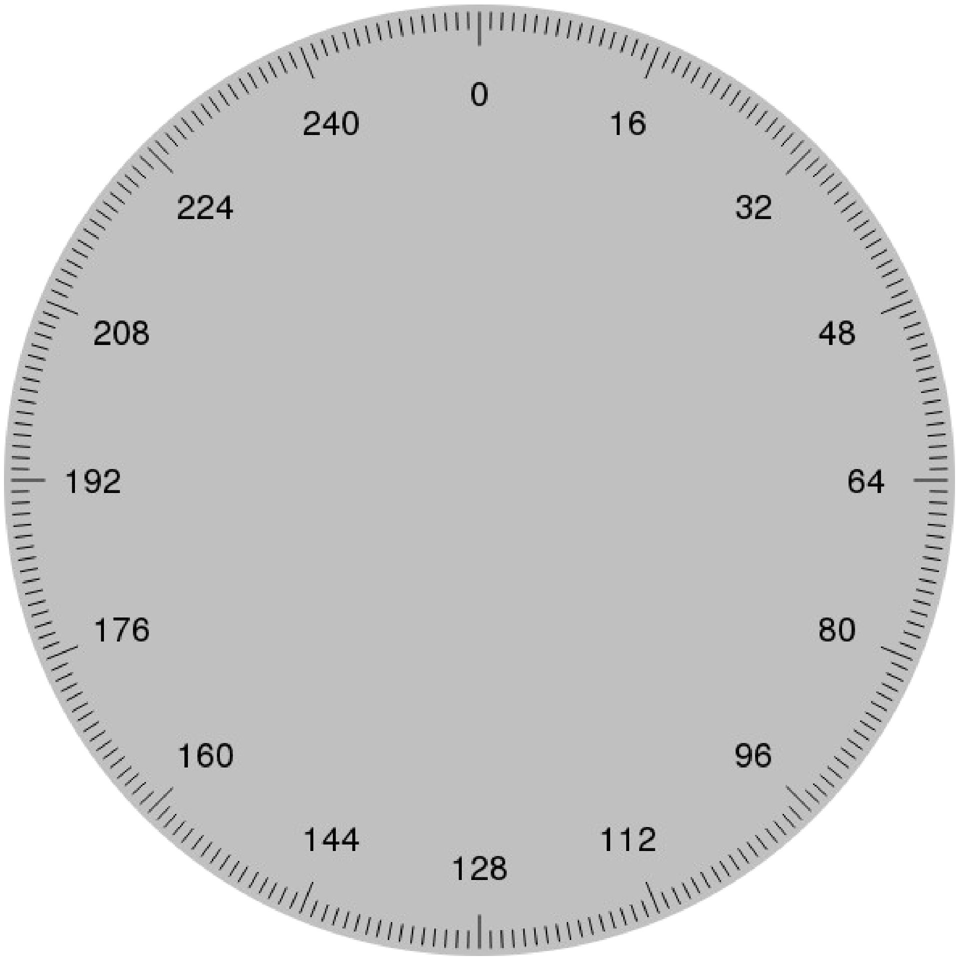
(element of $\mathbb{Z}/n$)

`unsigned size_t` was a mistake



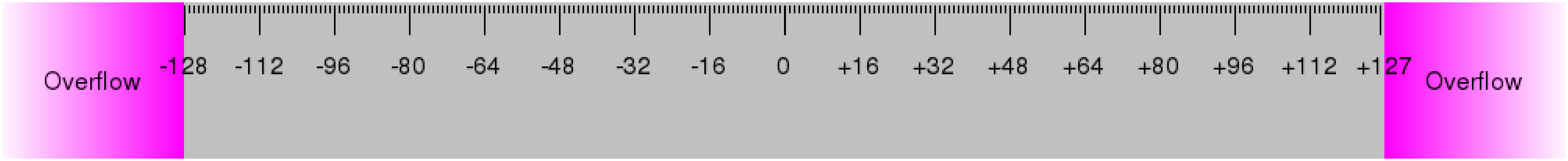
https://en.wikipedia.org/wiki/Modular_arithmetic

`unsigned size_t` was a mistake



`unsigned char` (8 bit)

`unsigned size_t` was a mistake



`signed char` (8 bit)

`unsigned size_t` was a mistake

Why is wrong using **`unsigned`** for **`size_t`**?

`unsigned size_t` was a mistake

Why is wrong using `unsigned` for `size_t`?

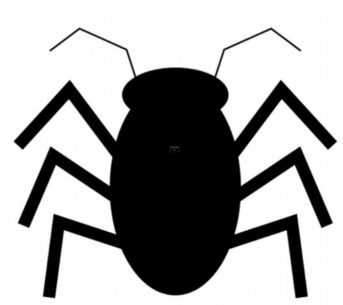
- `size_t` was meant to be the “size” of something
- “size” is conceptually a non-negative number
- `unsigned` is instead a modulo integer

`unsigned size_t` was a mistake

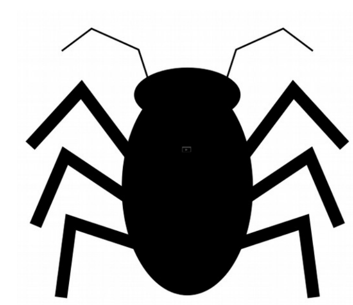
Why is wrong using `unsigned` for `size_t` for someone that *doesn’t care about philosophy*?

`unsigned size_t` was a mistake

Why is wrong using **`unsigned`** for **`size_t`** for someone that *doesn't care about philosophy*?



BUGS



`unsigned size_t` was a mistake

```
for (int i=0; i<pts.size()-1; i++){
    drawLine(pts[i], pts[i+1]);
}
```

`unsigned size_t` was a mistake

```
for (int i=0; i<pts.size()-1; i++){
    drawLine(pts[i], pts[i+1]);
}
```

When `pts` is empty this code is UB
(probably segfault)

`unsigned size_t` was a mistake

```
for (int i=0; i<pts.size()-1; i++){
    drawLine(pts[i], pts[i+1]);
}
```

Using **`size_t`** instead of **`int`** for index **`i`** would **NOT** solve the issue.

`unsigned size_t` was a mistake

```
for (int i=0; i<pts.size()-1; i++){
    drawLine(pts[i], pts[i+1]);
}
```

Using `size_t` instead of `int` for index `i` would **NOT** solve the issue.

It would just shut up the warning.

`unsigned size_t` was a mistake

```
for (int i=0; i<pts.size()-1; i++){
    drawLine(pts[i], pts[i+1]);
}
```

The problem is `pts.size()-1`,
not the index.
The problem is `0-1` = `4294967295`

`unsigned size_t` was a mistake

```
for (int i=0; i<pts.size()-1; i++){
    drawLine(pts[i], pts[i+1]);
}
```

A fix could be the use of

`…; i+1<pts.size();…`

instead of

`…; i<pts.size()-1;…`

`unsigned size_t` was a mistake

```
for (int i=0; i<pts.size()-1; i++){
    drawLine(pts[i], pts[i+1]);
}
```

When working with `unsigned` types

`A<B-1` is **NOT** the same as `A+1<B`

even for very common values like 0.

`unsigned size_t` was a mistake

```
for (int i=0,n=pts.size(); i<n-1; i++){
    drawLine(pts[i], pts[i+1]);
}
```

My *personally preferred* approach is to just get rid of `unsigned` types as soon as possible, and work with plain `int`.

`unsigned size_t` was a mistake

What are **`unsigned`** types good for?

A) If you actually *need* the modular arithmetic (e.g. cryptography) and you understand the implications

B) If you need to *use all the bits* explicitly (e.g. $\mathbf{b_7}$ **= `(1<<7) = 128`**, but as value is too big for a 8-bit **`signed char`**)

`unsigned size_t` was a mistake

“The unsigned integer types are ideal for uses that treat storage as a bit array. Using an unsigned instead of an int to gain one more bit to represent positive integers is almost never a good idea. Attempts to ensure that some values are positive by declaring variables unsigned will typically be defeated by the implicit conversion rules.”

Bjarne Stroustrup

`unsigned size_t` was a mistake

Using an `unsigned` type for `size_t` for standard containers size was a design mistake. The price to pay (wrong semantics) was too high for the little gain (one extra bit).

Unfortunately it cannot be fixed now because of backward compatibility.

`unsigned size_t` was a mistake

What *<u>can</u>* be done is *<u>avoid repeating</u>* the same mistake again in the future.

When designing new classes or new API please don't be fooled by the *name* into thinking that **`unsigned`** means non-negative: for the C++ language **`unsigned`** means modulo, or member of $\mathbb{Z}/n$.