What are unsigned integers?

Sometimes you'll see another integer type in C++ code, "unsigned" integer, that cannot represent negative values. Instead, unsigned integers have an increased upper positive value range compared to signed integers of the same memory usage. (Signed integers devote a bit of memory just to be able to represent a negative sign.) Unsigned integers are sometimes used in special cases to make use of memory extremely efficiently; for example, a data storage format might use them in order to be more compact. But mixing unsigned and signed integers in your code can cause unexpected problems. Let's walk through a few examples with unsigned integers.

You can download a source code version of this reading here:

unsigned-ints-example.zip

Unsigned type syntax

The normal "int" syntax creates a signed int by default. But, you could also write "signed int" or "signed" to get the same type:

```
int a = 7;
// signed int a = 7;
// signed a = 7;

```

If you write "unsigned int" or "unsigned", you can create a variable with the unsigned integer type:

```
unsigned int b = 8;
// unsigned b = 8;

```

Issues with unsigned arithmetic

Unsigned ints can't represent negative values. Instead, they sacrifice the ability to represent negative numbers in exchange for a higher upper positive limit to the value range that they can represent, using the same number of bits as the comparable signed integer. However, the underlying bit representation that unsigned integers use for these very large values is actually the same as the representation that signed integers use for their negative value range. **This means that a negative signed int may be re-interpreted as a very large positive unsigned int, and vice versa.** This can cause strange behavior if you mix signed and unsigned ints. If you do arithmetic between signed and unsigned ints, then you can get undesired results if negative values should logically have arisen.

Issues with addition

Addition with unsigned values may be no problem, as long as you don't exceed the maximum range. This is 7+8 and shows 15 as expected

```
std::cout << (a+b) << std::endl;

```

However, you do need to be careful if you are approaching the upper limit for a signed int even if you are using unsigned ints. Signed ints have a lower maximum value, so if you get an unsigned int greater than that limit and then cast it to signed int, it will be interpreted as a negative number you did not expect.

Issues with subtraction

Now let's look at issues that can arise if you do subtraction with unsigned ints and try to imply negative values that can't be represented. We'll create some unsigned ints:

```
unsigned int x = 10;
unsigned int y = 20;

```

Now writing (y-x) is 20-10 in unsigned arithmetic, which results in 10 as expected:

```
std::cout << (y-x) << std::endl;

```

By contrast, the next example attempts (x-y) which is 10-20, but the unsigned arithmetic can't represent -10, and instead results in a very large positive value. The output is 4294967286, which is close to the maximum for an unsigned 32-bit integer.

```
std::cout << (x-y) << std::endl;

```

Casting values back to signed int

If we explicitly cast the result back to a signed integer, then we might get something usable again. The output of this is -10:

```
std::cout << (int)(x-y) << std::endl;

```

You can also do a casting operation to convert to signed int just by assigning an unsigned result to a signed int variable. This also outputs -10:

```
int z = x - y;
std::cout << z << std::endl;

```

However, casting a result is not always the best way to handle this! Instead, you may want to create temporary working copies of unsigned values as signed types. That way, you can do operations as usual and manually check whether a value is negative or positive, in those cases where it should not be negative. For example, if you are assigning an unsigned int value to a signed int, you can check whether the signed int shows a negative value. If it is negative, that means the unsigned value was too large to be accurately cast to signed. Here's a contrived example:

```
int test_val = (x-y);
if (test_val < 0) {
  // Note: The standard error stream (cerr) will be displayed
  // You can also handle it separately from the standard outpu
  // if you are logging things to files.
  std::cerr << "Warning: unsigned value cast to signed int res
}

```

Making direct comparisons between signed and unsigned ints can also cause issues. This next line may give a warning or error. We have commented it out for now:

```
// std::cout << (a<b) << std::endl;

```

Container sizes are often unsigned

We often refer to a generic data structure class as a "container". The C++ Standard Template Library (STL) provides many of these, such as std::vector. Many of these class types have a size() member variable that returns the number of items the container currently holds. It's important to note that in many cases, this will give you the size in an unsigned integer type. (The exact byte size of the specific unsigned integer type may vary.) So, although you should probably try to avoid using unsigned integers in your code except in very special circumstances, you will run into cases where you are comparing a signed int (perhaps an iteration counter) with an unsigned integer size. So, be prepared to safely handle unsigned ints!

Here is an example where the compiler will warn you it's comparing the signed integer i with the unsigned integer returned by size():

```
std::vector<int> v = {1,2,3,4};
for (int i=0; i < v.size(); i++) {
    std::cout << v[i] << std::endl;
}

```

You could handle this using various casting methods described above to make the warning message go away. You could simply use an unsigned int, or indeed std::vector<int>::size_type itself, as the type for the counter i. (However, this may cause you more trouble if you are doing common arithmetic operations on i for one reason or another, because again, you are introducing an unsigned integer to your code. Let's suppose for now that your containers won't need to contain an astronomical number of items.)

Now, consider the danger of trying to *subtract* from the unsigned int that represents the size. If we wrote the code in the following way, what could go wrong?

```
std::vector<int> v = {1,2,3,4};
for (int i=0; i <= v.size()-1; i++) {
    std::cout << v[i] << std::endl;
}

```

This code seems to work as expected, but only because the size is nonzero. If we had any situation where this loop might process an empty vector, the program would crash:

```
std::vector<int> v;
for (int i=0; i <= v.size()-1; i++) {
    std::cout << v[i] << std::endl;
}

```

Here, the vector v is initialized empty by default, so its size() is reported as 0. That means the expression "v.size()-1" will evaluate to -1 *interpreted as an unsigned integer*, in this case meaning a very large positive integer. So, on the first loop iteration, "0 <= (a very large positive integer)" will be evaluated as true, and the loop body will execute, evaluating v[0], even though there is no first item in v to access. As a result, this code will most likely cause a segfault and crash.

So, here, it is most reasonable to write it the "i < v.size()" way perhaps, but these other tricks would also solve this issue. Let's think about why:

```
// Casting to signed int first helps to ensure that the result
// of subtraction will truly be a signed negative value when s
for (int i=0; i <= (int)v.size()-1; i++) {
    // ...
}

// Rewriting the algebra to perform addition instead of subtra
// helps to avoid going below 0:
for (int i=0; i+1 <= v.size(); i++) {
    // ...
}

```

In practice, for this code to actually be fully robust, you'd also need to check i against the maximum limit for an integer of its type, and make sure that the container never grew that large either. However, these are separate concerns. For now, just be sure to observe when you are converting to or from an unsigned integer type, and watch out for the potential problems it can cause.

Conclusion

Be careful when you are dealing with unsigned integers! In general, you should only use unsigned integers when you absolutely need to, such as if you are trying to maximize the range of positive values you can store in the same amount of memory. For everyday high-level programming purposes, signed integers may be all you need. If you need to ensure positive values during your code execution, you can write safety checks into the code to monitor values and issue warnings or errors if a certain range is exceeded.