BigInteger

CIS-3012, C++ Programming

Vermont State University

Peter Chapin

The Problem With BigInteger1

- Fixed maximum size
  - All BigInteger objects hold exactly 128 decimal digits
    - Can't expand beyond that when needed
    - Wastes memory for the (common) case when fewer digits are required
- Inefficient memory use
  - Each decimal digit consumes an entire int (32 bits in most cases) despite only containing $\log_2(10) = 3.32$ bits of information.

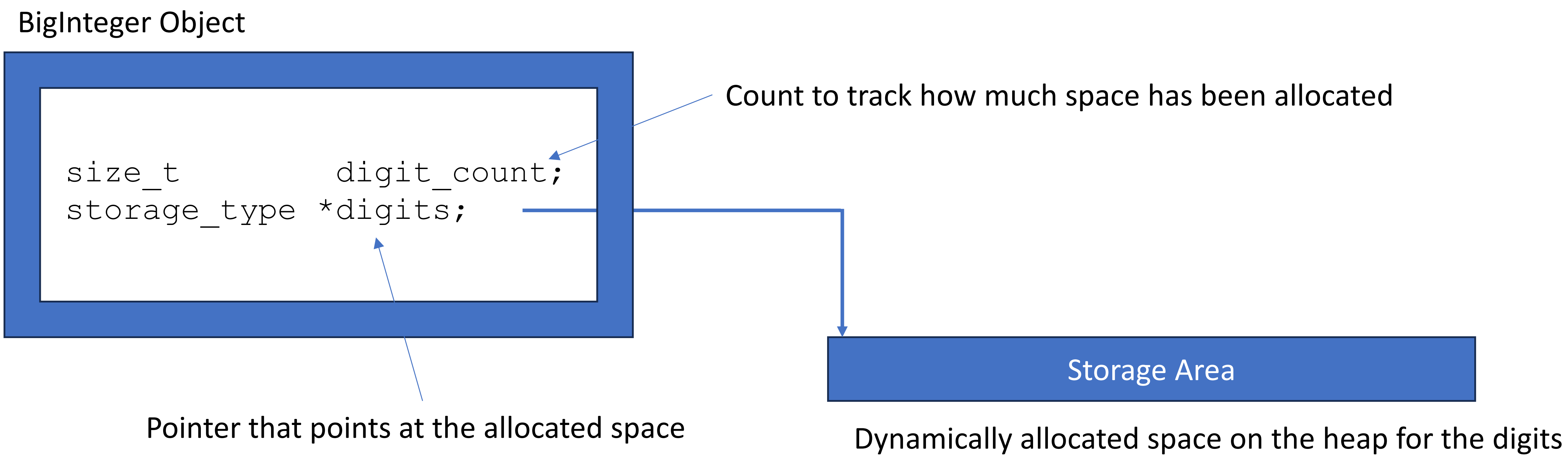
Dynamic Digits Array

- BigInteger2 will use a *dynamically allocated array* for the digits.
  - Each object can allocate whatever space it needs… but no more.
  - This allows object to hold potentially millions (billions!) of digits.
  - But small numbers won’t waste space.
- Different BigInteger objects will have different digit arrays…
  - ... with (in general) different sizes.
  - The size of the array is not part of the type.

What About `std::vector`?

- Aren't we supposed to use `std::vector` instead of arrays?
- Yes!
  - … but managing the allocations ourselves is <u>educational</u>.
  - It will help you understand how `std::vector`, and many other classes, are implemented.
- BigInteger4 will use vectors
  - You will see a huge simplification!

Object Layout



Notice that the BigInteger object is only 16 bytes (on 64-bit architectures)… regardless of how many digits are stored!
Most of the BigInteger's value is held externally to the object itself!

Now What?

- This is our first encounter with a class that holds most of its value externally.
- There are many implications!
  - How does the external data get released?
  - How does the external data get copied?
  - Isn’t it slow to copy potentially millions of bytes of data? How is that handled?
- BigInteger2 gives us a chance to address these questions
- Next up: Lifecycle Methods!
  - See the relevant slide deck for information on these special methods.

BigInteger3: Base 2^{32} Digits

- Instead of using base 10, we'll change to base 2^{32}
  - That is, our "digits" will be 32-bit unsigned numbers.
- Every bit is significant!
  - Very large values can be represented with a minimum number of total bits.
  - For example: a 32-million-bit number will be stored as one million 32-bit unsigned integers.
- Using a base which is a power of two…
  - … simplifies the math by allowing certain operations to be done as bit shifts and masking (very fast).

Compute Type vs Storage Type

- Computations on digits require temporaries with twice as many bits.
  - For example, in base 10, multiplying two 2-digit numbers yields a 4-digit result: 99 * 99 = 9801.
  - Similarly, multiplying two 32-bit numbers yields, in general, a 64-bit result.
- We will use two type aliases:
  - `storage_type` for holding a digit in memory (32 bits).
  - `compute_type` for holding temporary results of digit computation (64 bits).
  - Using type aliases improves code readability and documentation and allows us to modify their definition for different platforms.

What Platforms?

- First, who cares about arbitrary precision integers?
  - A classic use-case is cryptography. Some cryptographic algorithms manipulate (and do arithmetic on) values with thousands of bits (e.g., RSA, DSA, ECC).
- Will systems targeting microcontrollers ever want to use cryptography?
  - **Yes!**
  - So, it makes sense to ensure our code will work correctly even on a 16-bit processor. This requires attention to detail regarding the selection of integers types.
  - Hence the use of type aliases to make changes easy in the future.