A Crash Course on Data Compression

2.1 Some Integer Codes in C++

Giulio Ermanno Pibiri

ISTI-CNR, giulio.ermanno.pibiri@isti.cnr.it

@giulio_pibiri

@jermp

Overview

- C++ implementation of:
  Unary, Gamma, Delta, Rice, Variable-Byte

- Compress/Uncompress some (long) integer lists

- Write-to/Load-from disk the compressed file

General Approach

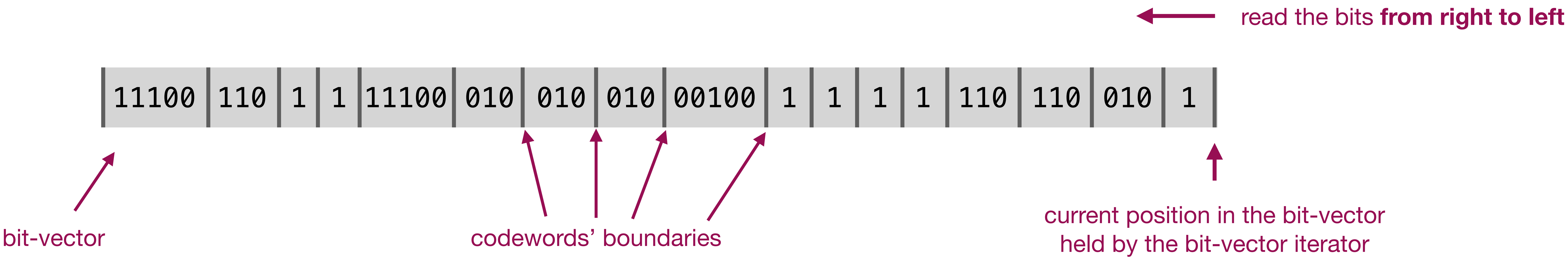
For the wanted code, implement **two** functions:

```
void write_codeword(bit_vector_builder& builder, uint64_t x)
```

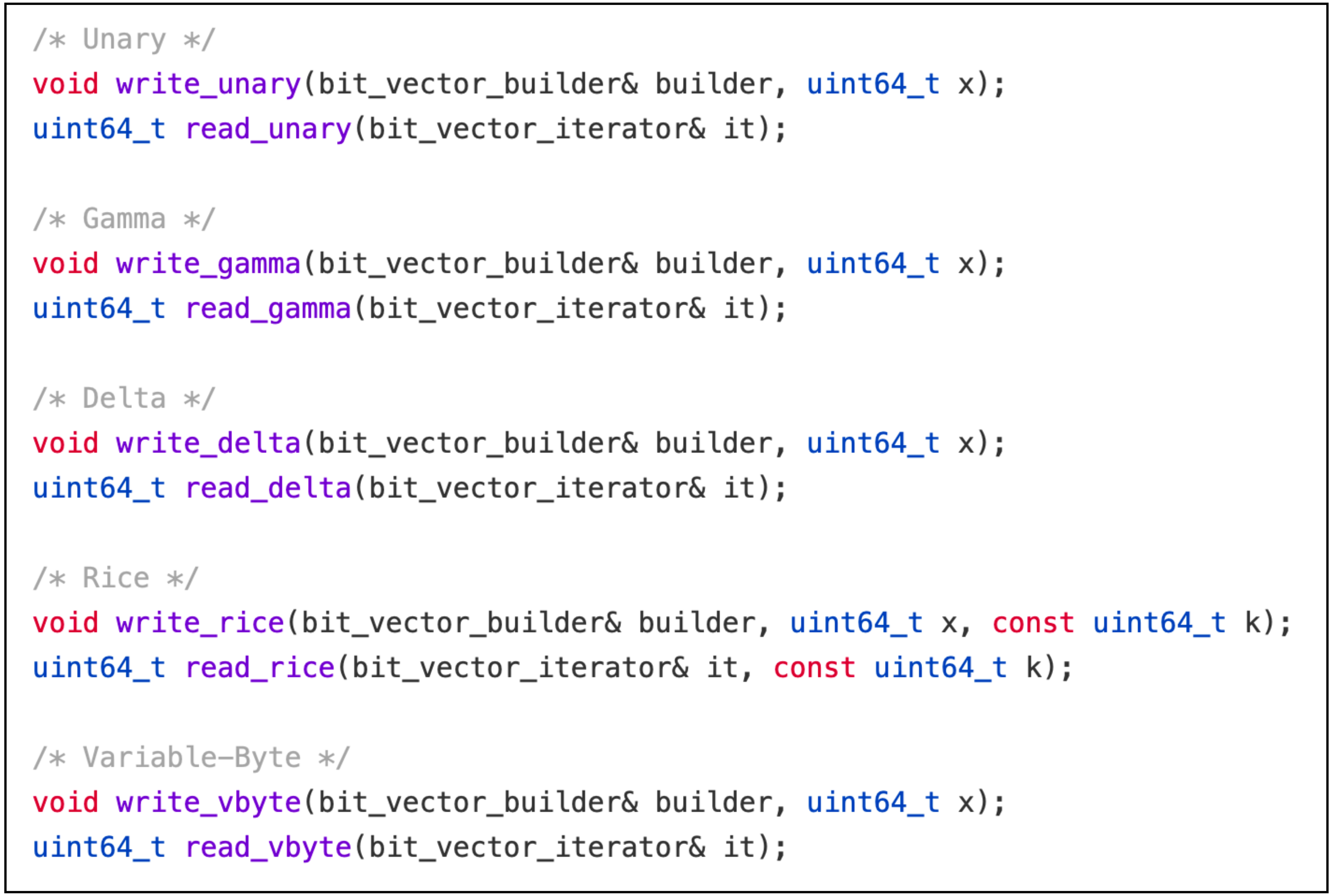
which appends the codeword for the integer x to the bit-vector builder's bits, and

```
uint64_t read_codeword(bit_vector_iterator& it)
```

which decodes the codeword coming next the current position of the bit-vector's iterator.



Our Plan



```
/* Unary */
void write_unary(bit_vector_builder& builder, uint64_t x);
uint64_t read_unary(bit_vector_iterator& it);

/* Gamma */
void write_gamma(bit_vector_builder& builder, uint64_t x);
uint64_t read_gamma(bit_vector_iterator& it);

/* Delta */
void write_delta(bit_vector_builder& builder, uint64_t x);
uint64_t read_delta(bit_vector_iterator& it);

/* Rice */
void write_rice(bit_vector_builder& builder, uint64_t x, const uint64_t k);
uint64_t read_rice(bit_vector_iterator& it, const uint64_t k);

/* Variable-Byte */
void write_vbyte(bit_vector_builder& builder, uint64_t x);
uint64_t read_vbyte(bit_vector_iterator& it);
```

Our Plan

All you (essentially) need to implement our plan is:

the function

```
void bit_vector_builder::append_bits(uint64_t x, uint64_t len)
```

appends the `len` **least significant** bits of x to the current end of the bit-vector

to implement `write_codeword`, and

the function

```
uint64_t bit_vector_iterator::take(uint64_t len)
```

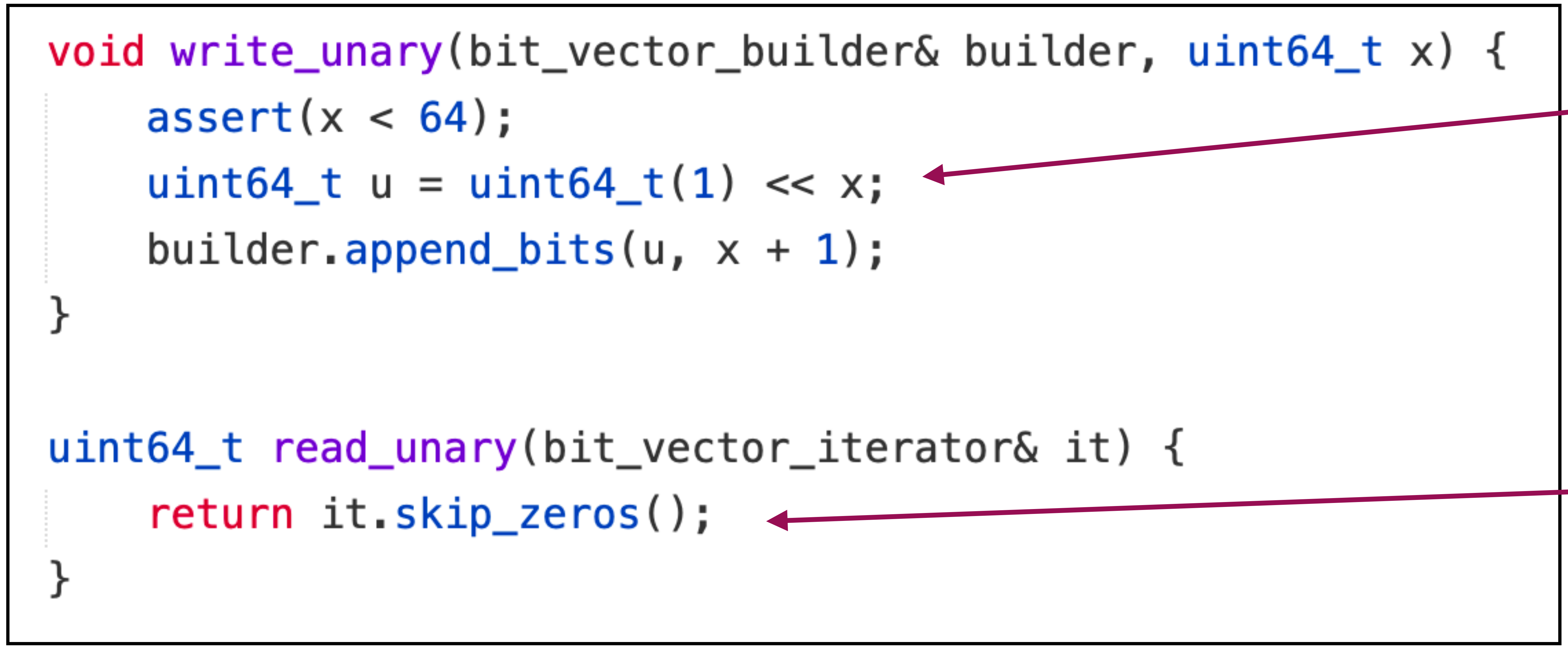
reads the next `len` bits from the bit-vector and returns them as the integer x

to implement `read_codeword`.

Unary

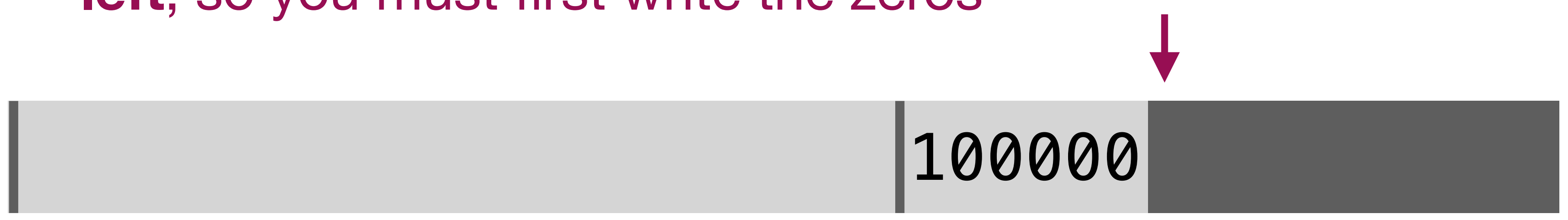
Represent $x \geq 0$ as $U(x) = 0^x 1$.

0	1
1	01
2	001
3	0001
…	



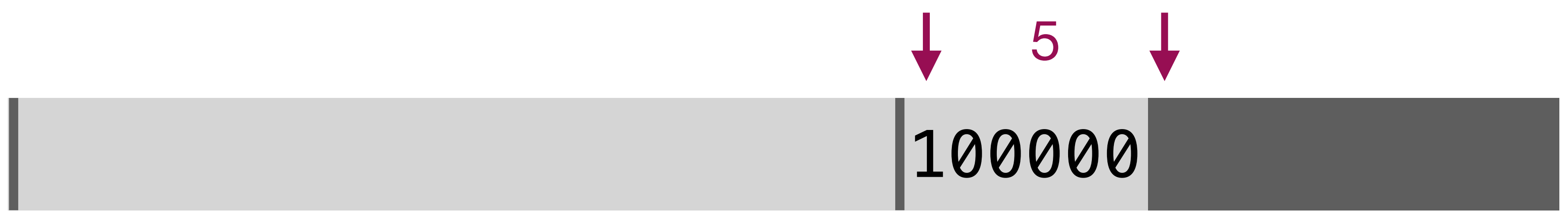
```
void write_unary(bit_vector_builder& builder, uint64_t x) {
    assert(x < 64);
    uint64_t u = uint64_t(1) << x;
    builder.append_bits(u, x + 1);
}

uint64_t read_unary(bit_vector_iterator& it) {
    return it.skip_zeros();
}
```

remember: you read the bits **from right to left**, so you must first write the zeros



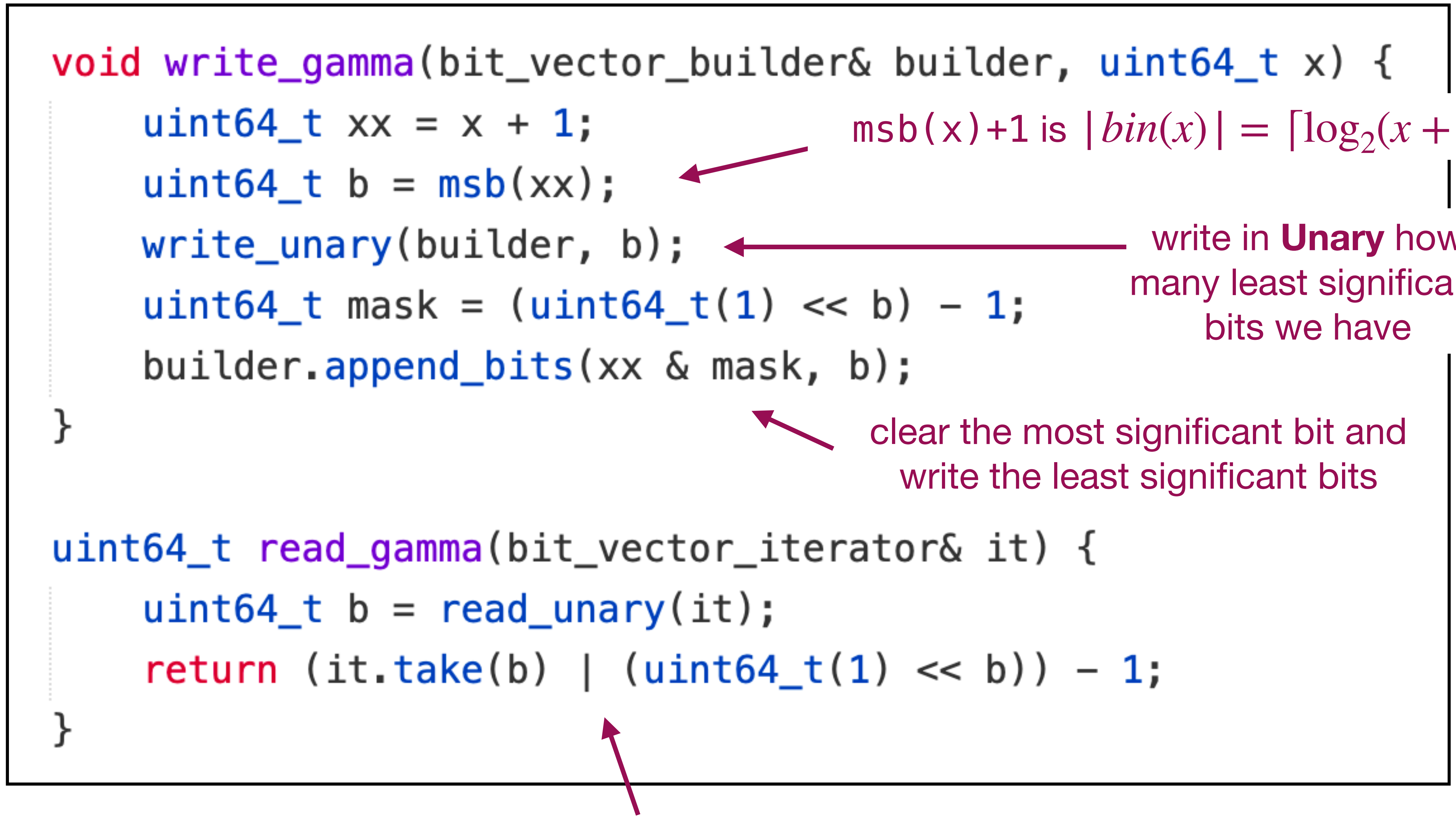
skip as many zeros as possible from the current position and return the number of skipped zeros

current position



Gamma

Write $b = |bin(x)| - 1$ using **Unary**, followed by the b least significant bits of $bin(x)$.



```
void write_gamma(bit_vector_builder& builder, uint64_t x) {
    uint64_t xx = x + 1;
    uint64_t b = msb(xx);
    write_unary(builder, b);
    uint64_t mask = (uint64_t(1) << b) - 1;
    builder.append_bits(xx & mask, b);
}

uint64_t read_gamma(bit_vector_iterator& it) {
    uint64_t b = read_unary(it);
    return (it.take(b) | (uint64_t(1) << b)) - 1;
}
```

`msb(x)+1` is $|bin(x)| = \lceil \log_2(x+1) \rceil$

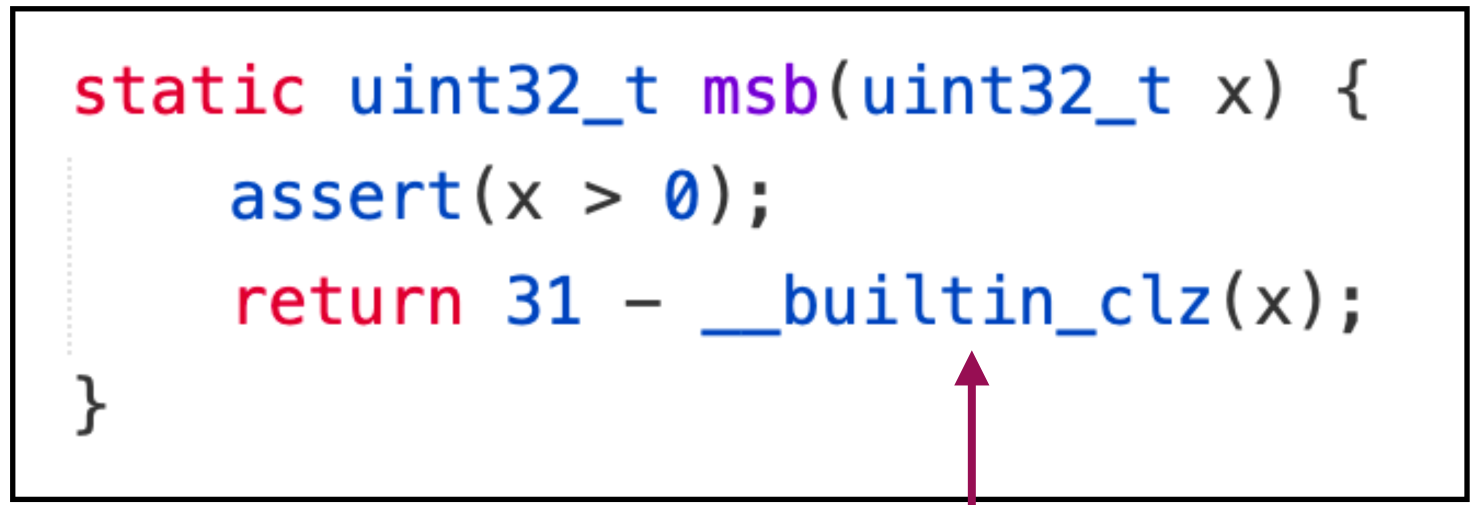
write in **Unary** how many least significant bits we have

clear the most significant bit and write the least significant bits

read the least significant bits and add the most significant bit

position of the most significant bit (msb)

```
00000000000000010101010101100111
00000000000000000001010100000101
```



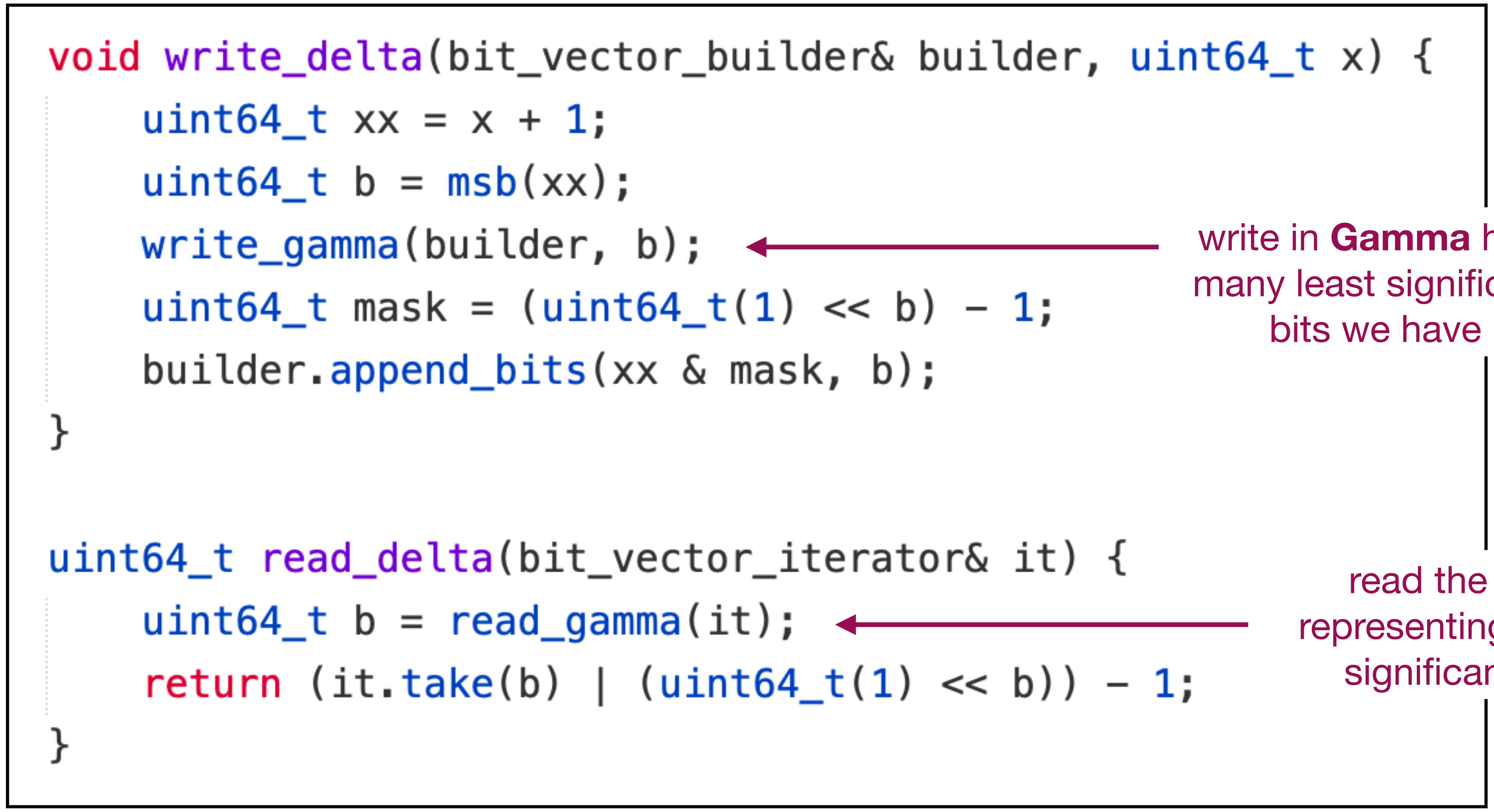
```
static uint32_t msb(uint32_t x) {
    assert(x > 0);
    return 31 - __builtin_clz(x);
}
```

count the number of leading zeros (clz) in a `uint32_t` word

```
00000000000000010101010101100111
00000000000000000001010100000101
```

Delta

Write $b = |bin(x)| - 1$ using **Gamma**, followed by the b least significant bits of $bin(x)$.



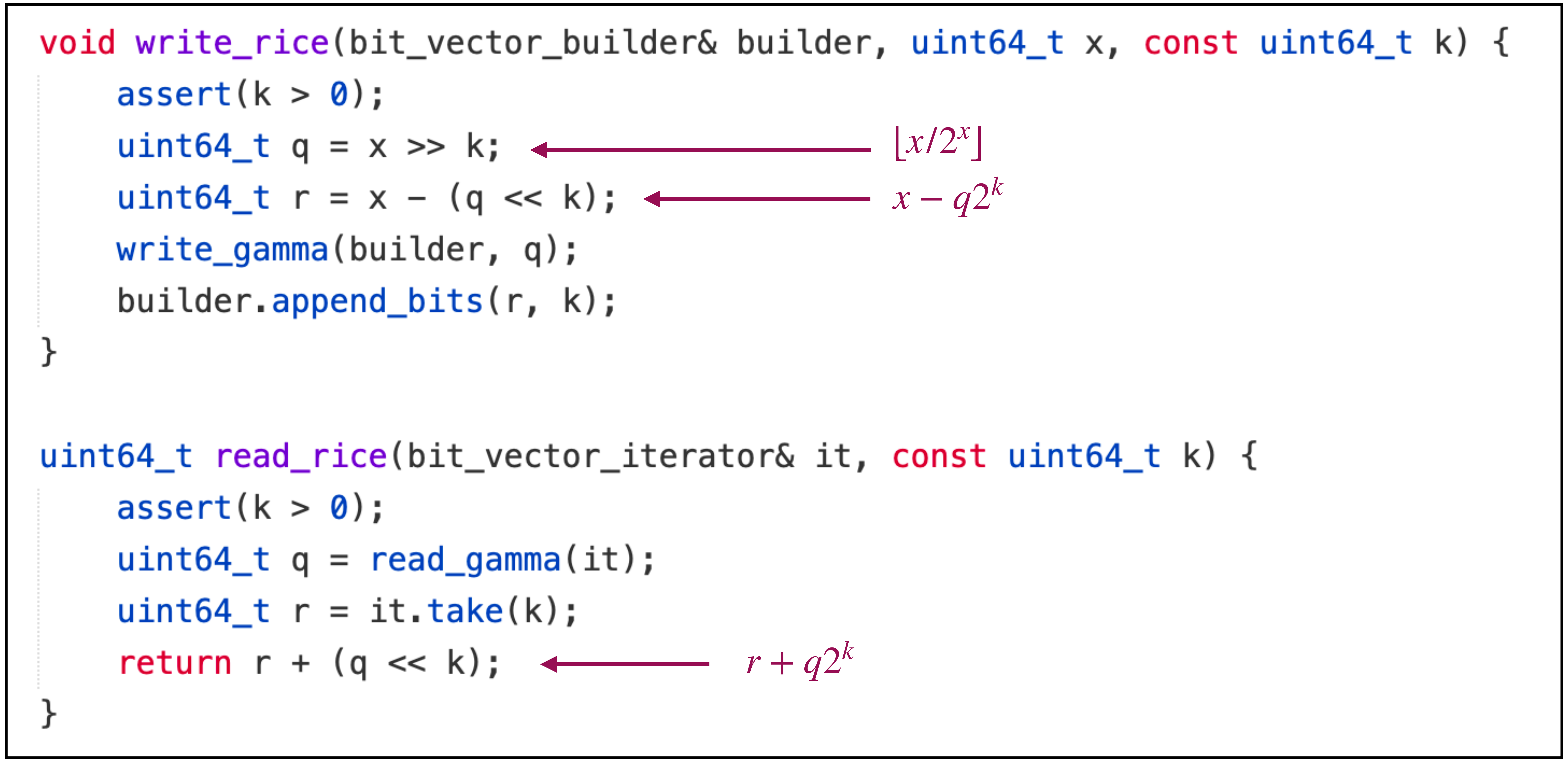
```
void write_delta(bit_vector_builder& builder, uint64_t x) {
    uint64_t xx = x + 1;
    uint64_t b = msb(xx);
    write_gamma(builder, b);
    uint64_t mask = (uint64_t(1) << b) - 1;
    builder.append_bits(xx & mask, b);
}

uint64_t read_delta(bit_vector_iterator& it) {
    uint64_t b = read_gamma(it);
    return (it.take(b) | (uint64_t(1) << b)) - 1;
}
```

write in **Gamma** how many least significant bits we have

read the **Gamma** code representing how many least significant bits we have

Rice

For a given parameter $k > 0$, write $q = \lfloor x/2^k \rfloor$ using **Gamma** followed by the reminder $r = x - q2^k$ in k bits.



```
void write_rice(bit_vector_builder& builder, uint64_t x, const uint64_t k) {
    assert(k > 0);
    uint64_t q = x >> k;
    uint64_t r = x - (q << k);
    write_gamma(builder, q);
    builder.append_bits(r, k);
}

uint64_t read_rice(bit_vector_iterator& it, const uint64_t k) {
    assert(k > 0);
    uint64_t q = read_gamma(it);
    uint64_t r = it.take(k);
    return r + (q << k);
}
```

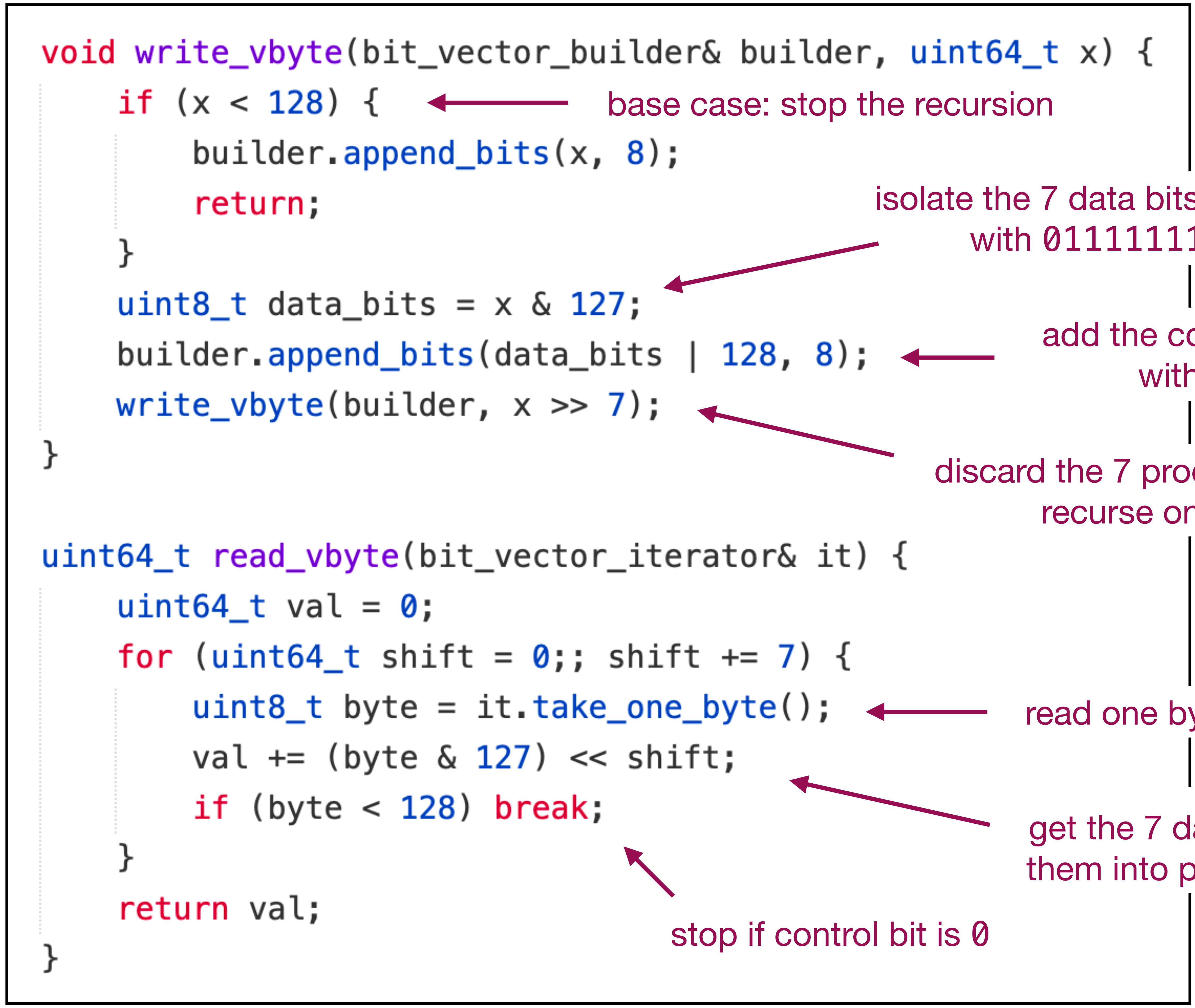
Variable-Byte

Example for $x = 67822$, $bin(67822,17)$ = `10000100011101110`.

`00000100.10010001.11101110`

recursive implementation
(for elegance)

iterative implementation
(for efficiency)



```
void write_vbyte(bit_vector_builder& builder, uint64_t x) {
    if (x < 128) {
        builder.append_bits(x, 8);
        return;
    }
    uint8_t data_bits = x & 127;
    builder.append_bits(data_bits | 128, 8);
    write_vbyte(builder, x >> 7);
}

uint64_t read_vbyte(bit_vector_iterator& it) {
    uint64_t val = 0;
    for (uint64_t shift = 0;; shift += 7) {
        uint8_t byte = it.take_one_byte();
        val += (byte & 127) << shift;
        if (byte < 128) break;
    }
    return val;
}
```

base case: stop the recursion

isolate the 7 data bits by masking
with `01111111` (127)

add the continuation bit by ORing
with `10000000` (128)

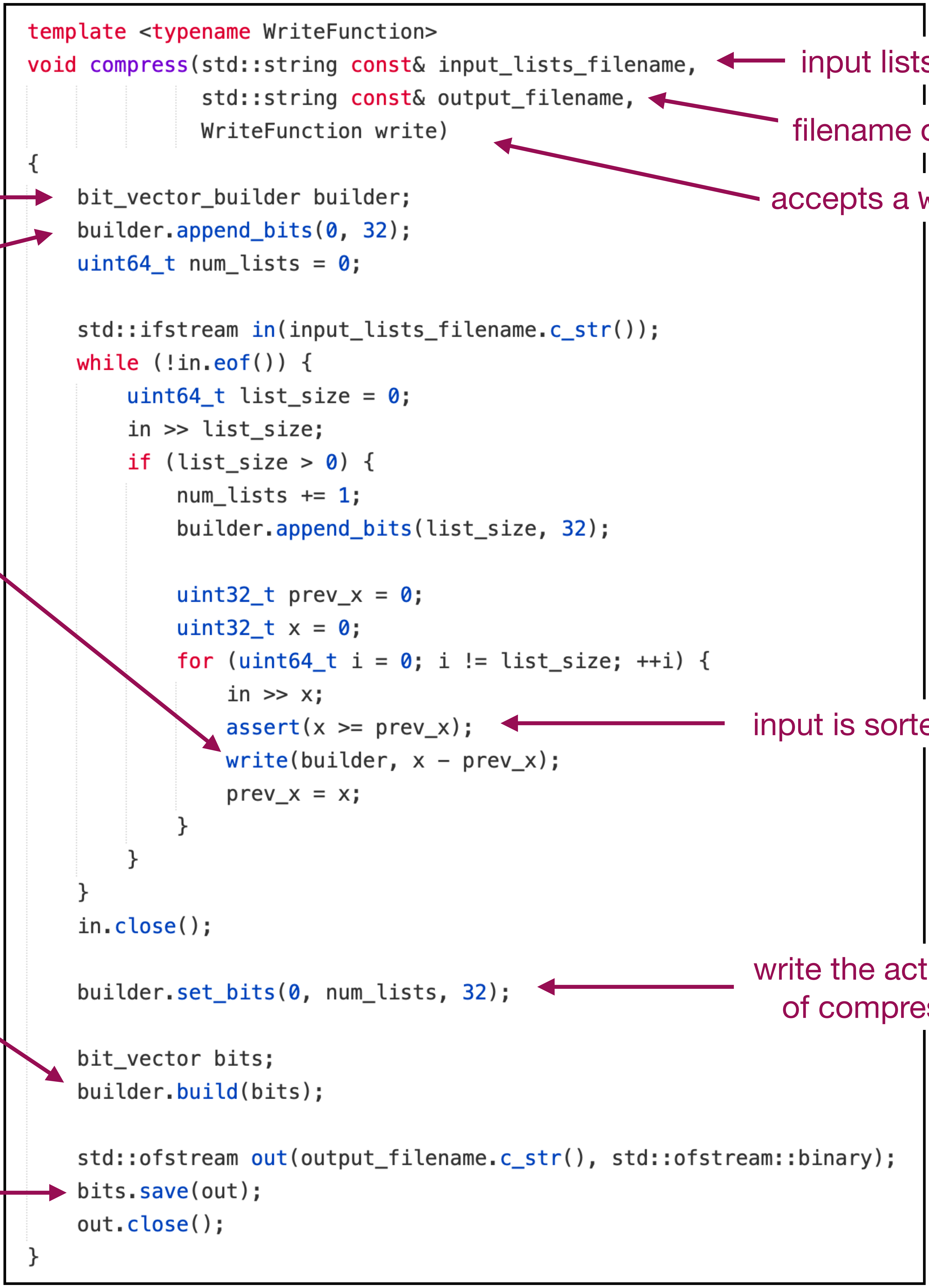
discard the 7 processed bits and
recurse on the rest

read one byte

get the 7 data bits and place
them into position by shifting

stop if control bit is `0`

Compress



```cpp
template <typename WriteFunction>
void compress(std::string const& input_lists_filename,
              std::string const& output_filename,
              WriteFunction write)
{
    bit_vector_builder builder;
    builder.append_bits(0, 32);
    uint64_t num_lists = 0;

    std::ifstream in(input_lists_filename.c_str());
    while (!in.eof()) {
        uint64_t list_size = 0;
        in >> list_size;
        if (list_size > 0) {
            num_lists += 1;
            builder.append_bits(list_size, 32);

            uint32_t prev_x = 0;
            uint32_t x = 0;
            for (uint64_t i = 0; i != list_size; ++i) {
                in >> x;
                assert(x >= prev_x);
                write(builder, x - prev_x);
                prev_x = x;
            }
        }
    }
    in.close();

    builder.set_bits(0, num_lists, 32);

    bit_vector bits;
    builder.build(bits);

    std::ofstream out(output_filename.c_str(), std::ofstream::binary);
    bits.save(out);
    out.close();
}
```

input lists' filename

filename of the compressed output

accepts a `write_codeword` function

create the bit-vector's builder

reserve space for the number of lists

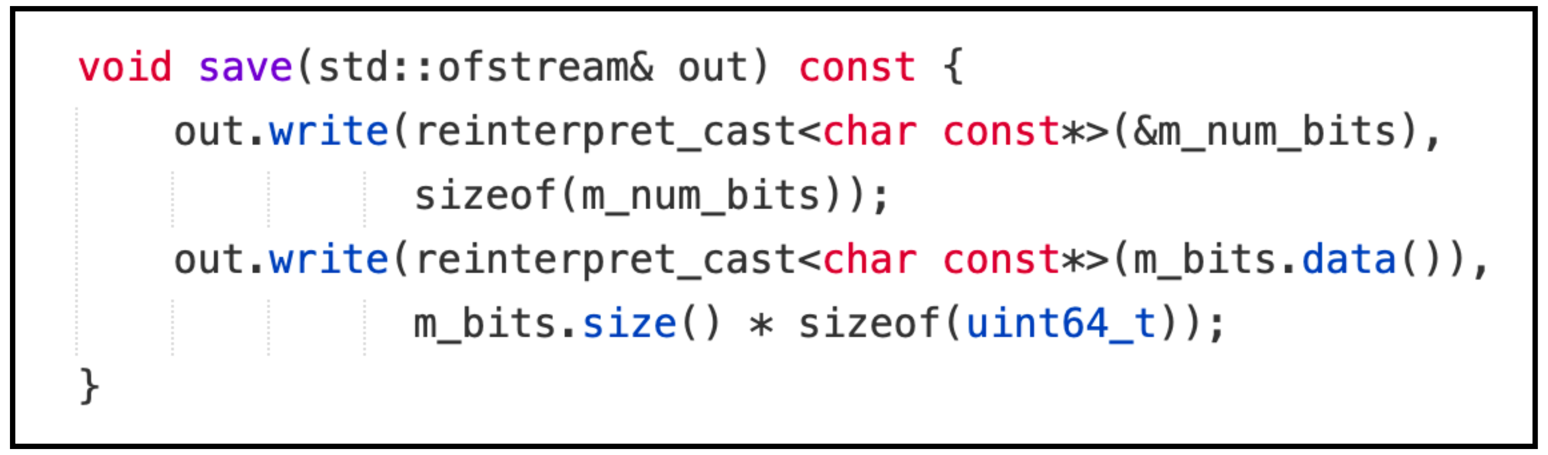
encode the gaps
between the integers

input is sorted

write the actual number
of compressed lists

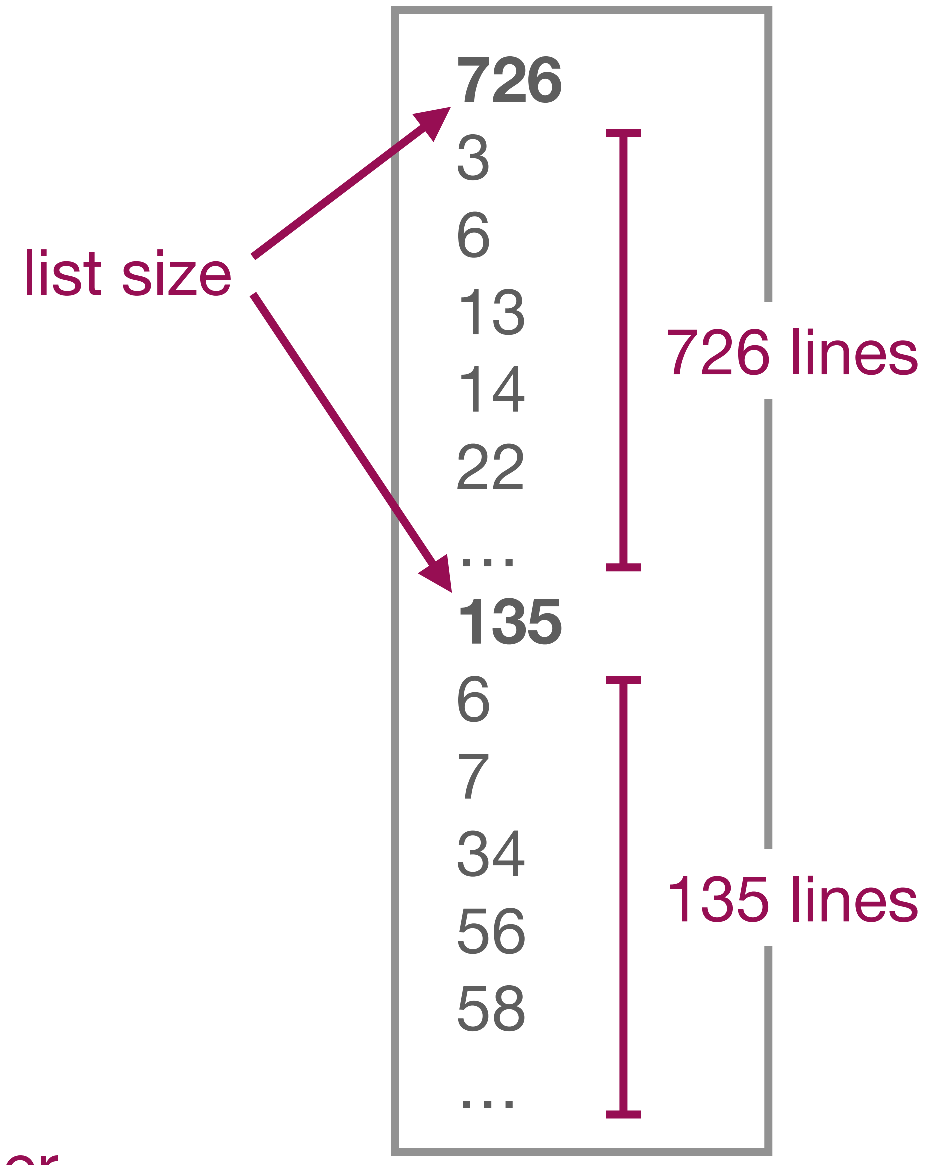
build the vector

write all the bytes
to the output file



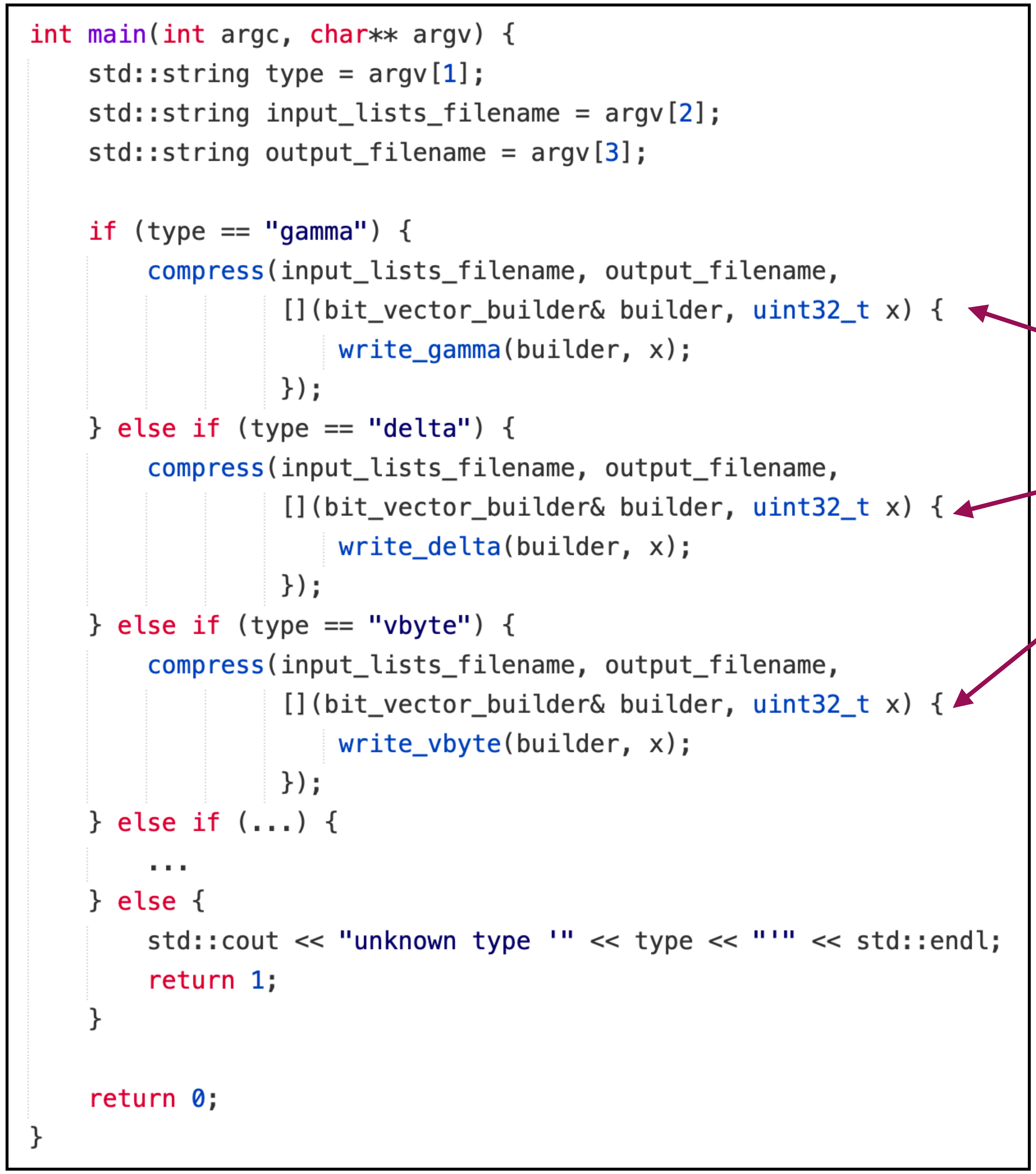
```cpp
void save(std::ofstream& out) const {
    out.write(reinterpret_cast<char const*>(&m_num_bits),
              sizeof(m_num_bits));
    out.write(reinterpret_cast<char const*>(m_bits.data()),
              m_bits.size() * sizeof(uint64_t));
}
```

example input: lists.txt



```
726
3
6
13
14
22
…
135
6
7
34
56
58
…
```

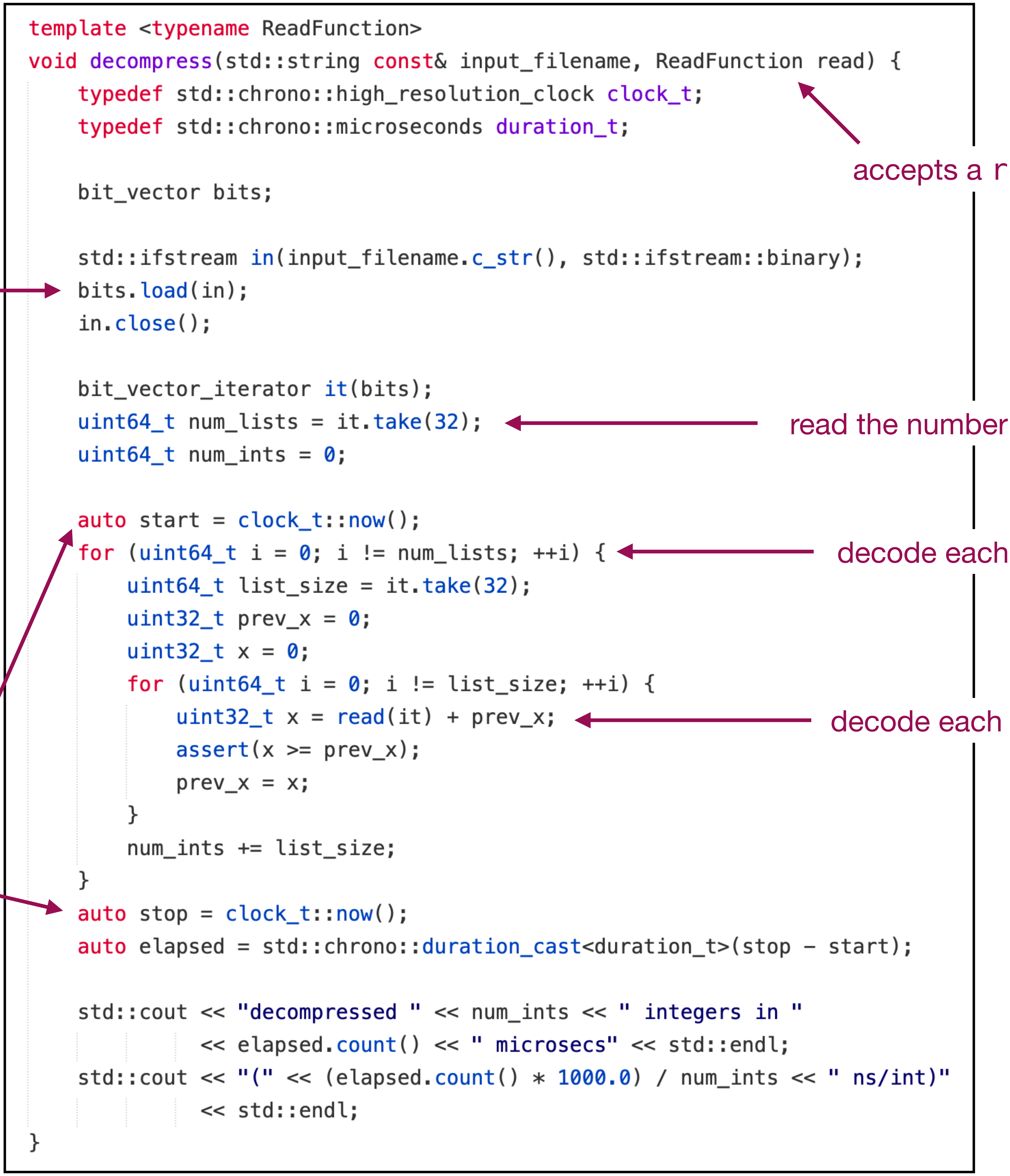
Compress



```cpp
int main(int argc, char** argv) {
    std::string type = argv[1];
    std::string input_lists_filename = argv[2];
    std::string output_filename = argv[3];

    if (type == "gamma") {
        compress(input_lists_filename, output_filename,
                 [](bit_vector_builder& builder, uint32_t x) {
                     write_gamma(builder, x);
                 });
    } else if (type == "delta") {
        compress(input_lists_filename, output_filename,
                 [](bit_vector_builder& builder, uint32_t x) {
                     write_delta(builder, x);
                 });
    } else if (type == "vbyte") {
        compress(input_lists_filename, output_filename,
                 [](bit_vector_builder& builder, uint32_t x) {
                     write_vbyte(builder, x);
                 });
    } else if (...) {
        ...
    } else {
        std::cout << "unknown type '" << type << "'" << std::endl;
        return 1;
    }

    return 0;
}
```

pass the specific `write_codeword` function

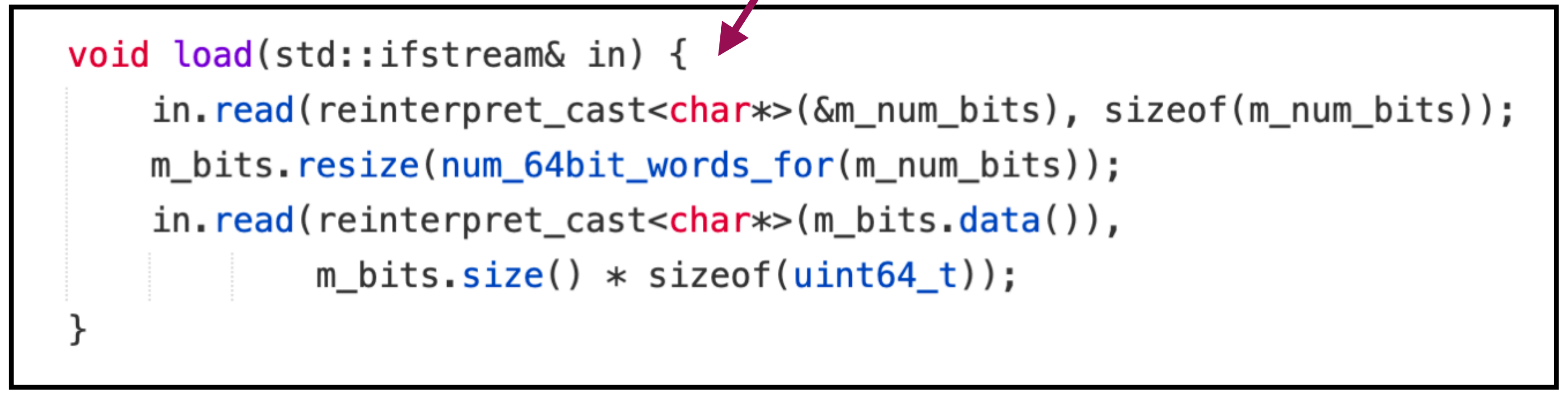
Decompress



```
template <typename ReadFunction>
void decompress(std::string const& input_filename, ReadFunction read) {
    typedef std::chrono::high_resolution_clock clock_t;
    typedef std::chrono::microseconds duration_t;

    bit_vector bits;

    std::ifstream in(input_filename.c_str(), std::ifstream::binary);
    bits.load(in);
    in.close();

    bit_vector_iterator it(bits);
    uint64_t num_lists = it.take(32);
    uint64_t num_ints = 0;

    auto start = clock_t::now();
    for (uint64_t i = 0; i != num_lists; ++i) {
        uint64_t list_size = it.take(32);
        uint32_t prev_x = 0;
        uint32_t x = 0;
        for (uint64_t i = 0; i != list_size; ++i) {
            uint32_t x = read(it) + prev_x;
            assert(x >= prev_x);
            prev_x = x;
        }
        num_ints += list_size;
    }
    auto stop = clock_t::now();
    auto elapsed = std::chrono::duration_cast<duration_t>(stop - start);

    std::cout << "decompressed " << num_ints << " integers in "
              << elapsed.count() << " microsecs" << std::endl;
    std::cout << "(" << (elapsed.count() * 1000.0) / num_ints << " ns/int)"
              << std::endl;
}
```

accepts a `read_codeword` function

read all the bytes from the file into memory



```
void load(std::ifstream& in) {
    in.read(reinterpret_cast<char*>(&m_num_bits), sizeof(m_num_bits));
    m_bits.resize(num_64bit_words_for(m_num_bits));
    in.read(reinterpret_cast<char*>(m_bits.data()),
            m_bits.size() * sizeof(uint64_t));
}
```

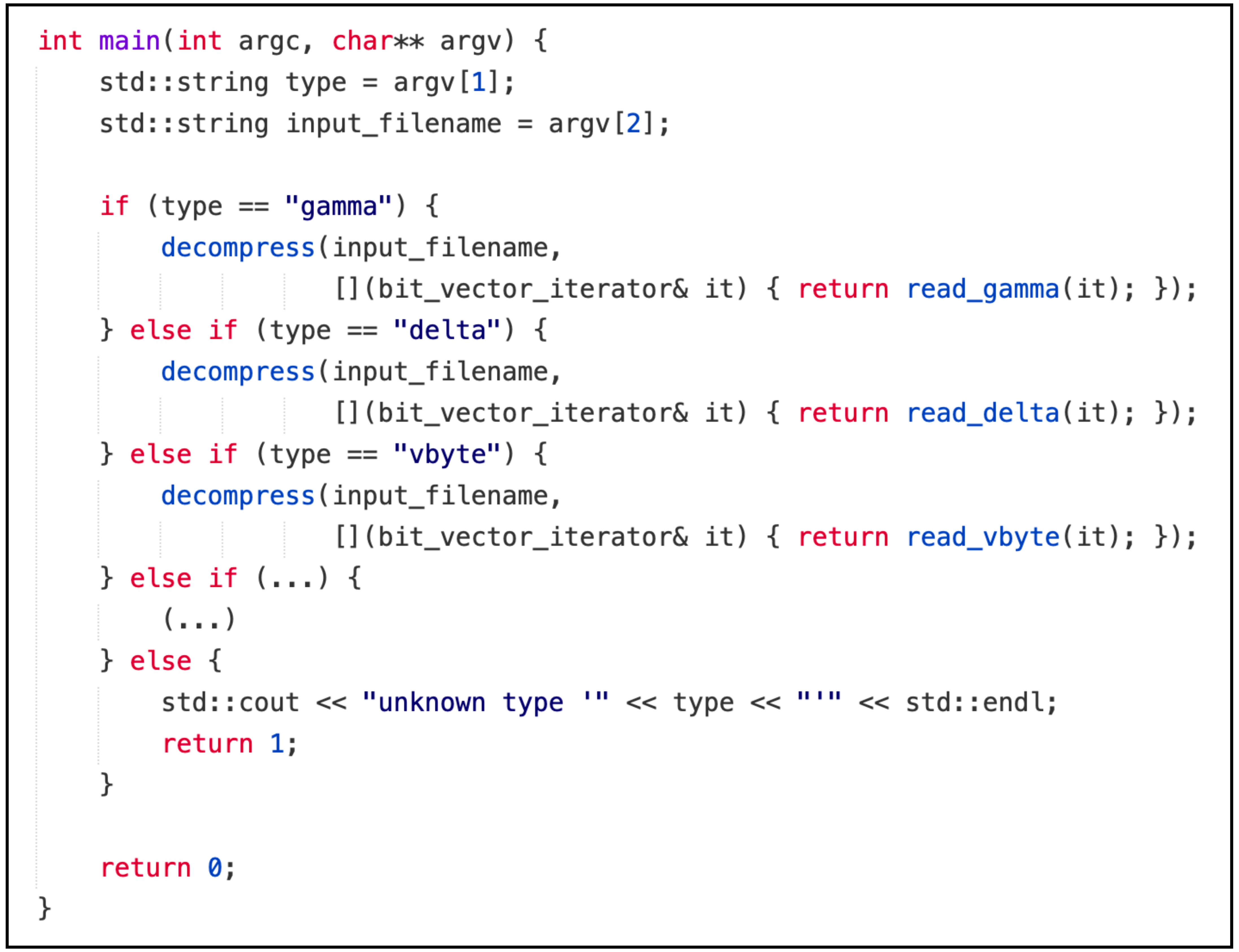
read the number of lists

decode each list

decode each gap

measure the decoding time using `std::chrono`

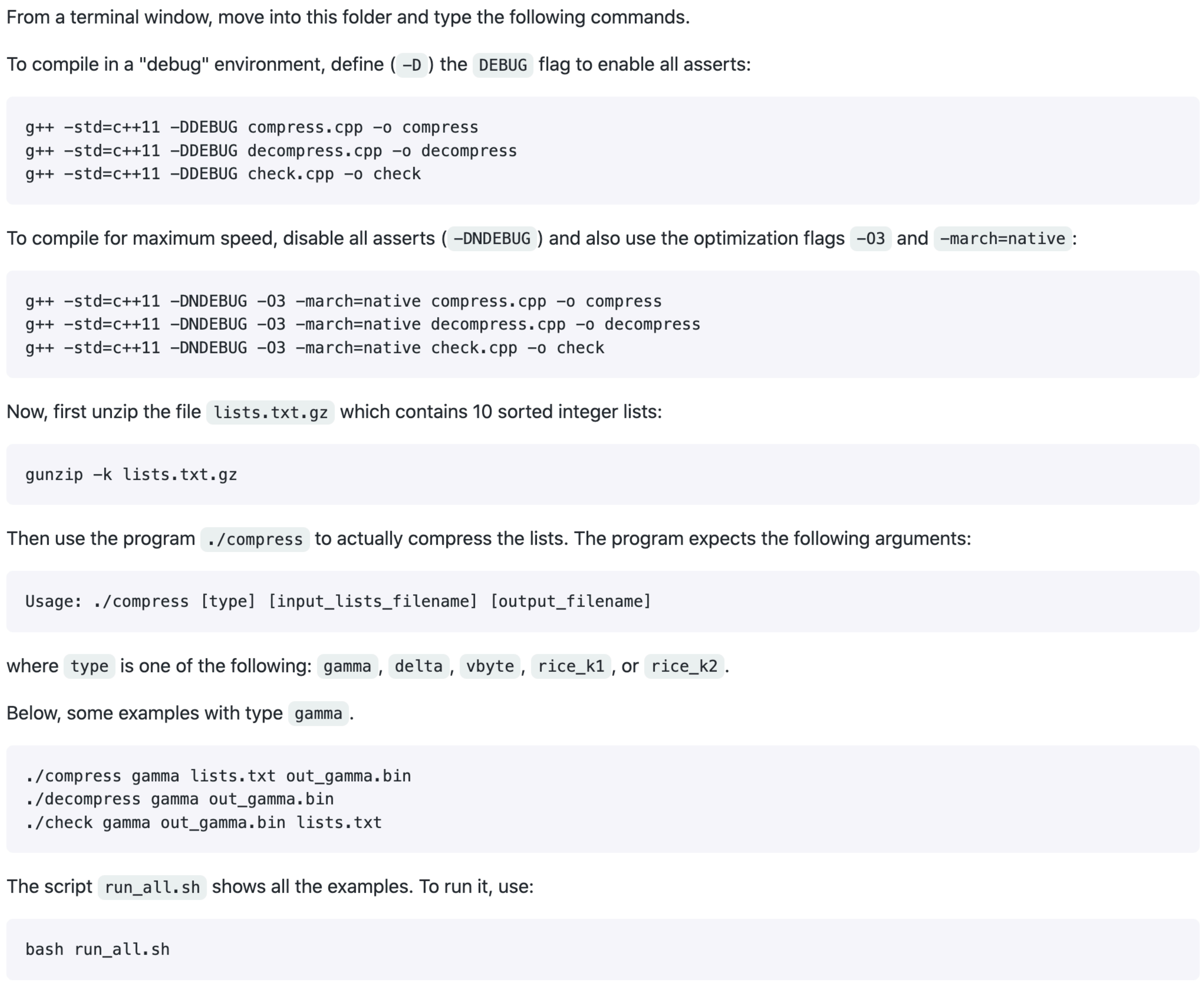
Decompress



```
int main(int argc, char** argv) {
    std::string type = argv[1];
    std::string input_filename = argv[2];

    if (type == "gamma") {
        decompress(input_filename,
                   [](bit_vector_iterator& it) { return read_gamma(it); });
    } else if (type == "delta") {
        decompress(input_filename,
                   [](bit_vector_iterator& it) { return read_delta(it); });
    } else if (type == "vbyte") {
        decompress(input_filename,
                   [](bit_vector_iterator& it) { return read_vbyte(it); });
    } else if (...) {
        (...)
    } else {
        std::cout << "unknown type '" << type << "'" << std::endl;
        return 1;
    }

    return 0;
}
```

pass the specific `read_codeword` function

How to compile and run the code



From a terminal window, move into this folder and type the following commands.

To compile in a "debug" environment, define (`-D`) the `DEBUG` flag to enable all asserts:

```
g++ -std=c++11 -DDEBUG compress.cpp -o compress
g++ -std=c++11 -DDEBUG decompress.cpp -o decompress
g++ -std=c++11 -DDEBUG check.cpp -o check
```

To compile for maximum speed, disable all asserts (`-DNDEBUG`) and also use the optimization flags `-O3` and `-march=native` :

```
g++ -std=c++11 -DNDEBUG -O3 -march=native compress.cpp -o compress
g++ -std=c++11 -DNDEBUG -O3 -march=native decompress.cpp -o decompress
g++ -std=c++11 -DNDEBUG -O3 -march=native check.cpp -o check
```

Now, first unzip the file `lists.txt.gz` which contains 10 sorted integer lists:

```
gunzip -k lists.txt.gz
```

Then use the program `./compress` to actually compress the lists. The program expects the following arguments:

```
Usage: ./compress [type] [input_lists_filename] [output_filename]
```

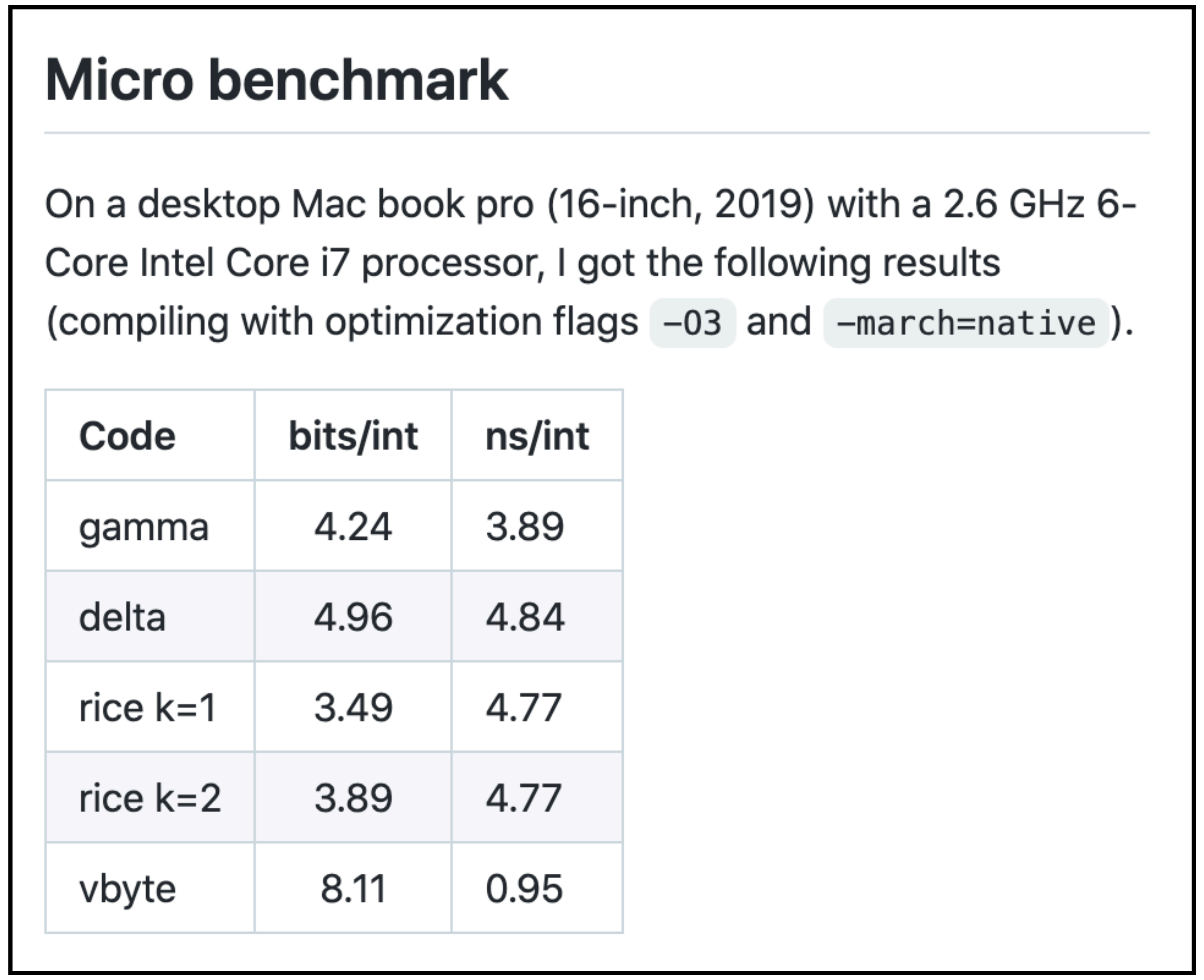
where `type` is one of the following: `gamma` , `delta` , `vbyte` , `rice_k1` , or `rice_k2` .

Below, some examples with type `gamma` .

```
./compress gamma lists.txt out_gamma.bin
./decompress gamma out_gamma.bin
./check gamma out_gamma.bin lists.txt
```

The script `run_all.sh` shows all the examples. To run it, use:

```
bash run_all.sh
```



Micro benchmark

On a desktop Mac book pro (16-inch, 2019) with a 2.6 GHz 6-Core Intel Core i7 processor, I got the following results (compiling with optimization flags `-O3` and `-march=native`).

Code	bits/int	ns/int
gamma	4.24	3.89
delta	4.96	4.84
rice k=1	3.49	4.77
rice k=2	3.89	4.77
vbyte	8.11	0.95