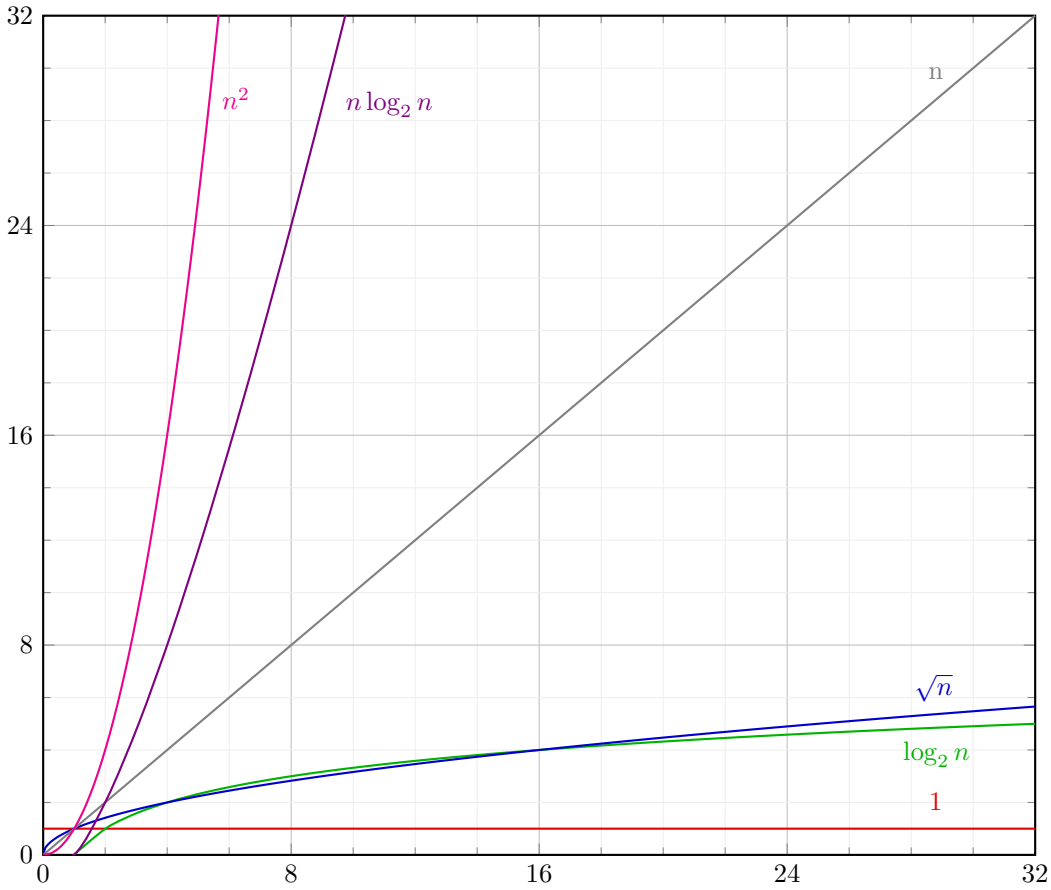
codex



Bruno Barcarol Guimarães

MMXXI

Contents

Chapter 1

Computer architecture

1.1 Integers

1.1.1 Binary numbers and modular arithmetic

Computers must operate on finite values, including the size of numbers they can represent. The *word size*[1]of an architecture determines the natural unit of data of the instruction set and limits the size of numbers. Early computers had 8-bit words, several word sizes have existed throughout history, but common sizes are now 32 and 64 bits.

This finiteness means operations will eventually exceed the minimum or maximum values — situations called *underflow* and *overflow*, respectively. One way to deal with that is through overflow exceptions[2], but that would be expensive and would mean certain mathematical properties would not be retained. `(x + x) - x` is no longer equal to `x + (x - x)` because the sum in the former can overflow, while the latter will always equal `x`.

Another option in the case of integers is to perform *modular arithmetic*: operations are carried out *modulo* the number of representable integers. A binary number of `n` bits (8, 16, 32, 64, etc.) has 2^n possible values; operations on integer values are thus *modulo* 2^n and preserve all ordinary properties of operations between integers.

$$
\begin{aligned}
x + y &\equiv_{\bmod n} y + x && \text{(additive commutativity)} \\
(x + y) + z &\equiv_{\bmod n} x + (y + z) && \text{(additive associativity)} \\
x + 0 &\equiv_{\bmod n} x && \text{(additive unit)} \\
\\
x + (-x) &\equiv_{\bmod n} 0 && \text{(additive inverse)} \\
-(-x) &\equiv_{\bmod n} x && \text{(cancellation)} \\
\\
x \times y &\equiv_{\bmod n} y \times x && \text{(multiplicative commutativity)} \\
(x \times y) \times z &\equiv_{\bmod n} x \times (y \times z) && \text{(multiplicative associativity)} \\
x \times 1 &\equiv_{\bmod n} x && \text{(multiplicative unit)} \\
\\
x \times (y + z) &\equiv_{\bmod n} x \times y + x \times z && \text{(distributivity)} \\
x \times 0 &\equiv_{\bmod n} 0 && \text{(annihilation)}
\end{aligned}
$$

[1]Used here *stricto sensu*, in contrast to the alternative definition of "word" as the word size of a particular ancestor of the architecture that is now maintained for backwards compatibility (e.g. "16-bit word" in x86).

[2]Meaning a *hardware* exception/fault in this case, not the similar software concept with the same name.

Addition

An addition can be defined in terms of its inputs — the *augend* and the *addend* — and its outputs — the *sum* and the *carry*. In the case of one-digit binary numbers with modular arithmetic, the sum is equivalent to an *xor* operation (also denoted as $\oplus$ for that reason). Similarly, the carry is equivalent to an *and* operation (also denoted as $\cdot$). The resulting expression, `s = (au + ad) % 2, c = au * ad`, can be represented in a truth table and encoded in a logic circuit (figure 1.1).

Input		Output	
AU	AD	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

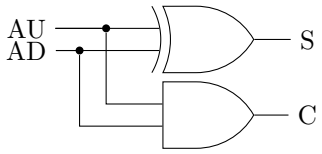


Figure 1.1: One-digit binary half adder

This circuit, a *half adder*, can be combined with an external carry input to form a *full adder* (figure 1.2). Multiple full adders can then be connected in sequence to produce an adder for an arbitrary number of digits, a *ripple-carry adder*[3]. In an *arithmetic logic unit* (ALU), the carry output of the last adder is connected to the *carry flag*, which can be used in instructions following the addition, including other additions, where it serves as the input carry to the first adder. One example of its usage is to implement addition of numbers larger than the maximum supported integer type: a pair of arrays of numbers can be summed in sequence, with each iteration adding the carry flag to the next (more significant) group of digits.

Input			Output	
AU	AD	C_{in}	S	C
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

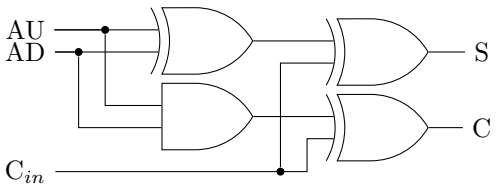


Figure 1.2: One-digit binary adder

Since integer operations are modulo 2^n, where n is the word size and the size of the ALU registers, simply truncating the result to the register size — i.e. ignoring the carry — effects modular arithmetic. Listing 1.1 shows examples of addition with 4-bit integers.

Subtraction

Subtraction is in many ways analogous to addition: a *minuend*, *subtrahend*, and a *borrow* are combined to produce the *difference* and the subsequent borrow for each bit and the last output borrow affects the carry flag. In fact, for one-digit binary numbers, the computation of the

[3] Alternative designs for the half/full adders allow faster implementations than a naive concatenation of adders.

```
 0b0100 = 4     0b0111 = 7       0b1100 = 12
+0b0011 = 3    +0b0001 = 1      +0b0111 =  7
-------        -------          -------
 0b0111 = 7     0b0110           0b1011
                   ^^  carry      ^^     carry
                -------          -------
                 0b1000 = 8      0b10011 = 19
                                   ^      carry flag
                                -------
                                 0b0011 = 19 % 16
                                        = 3
```

Listing 1.1: Addition with modular arithmetic

difference is exactly the same as that of the sum (figure 1.3). Computing the borrow also uses an *and* gate but requires an extra *not* gate. Figure 1.4 shows the full subtractor[4].

Input		Output	
M	S	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

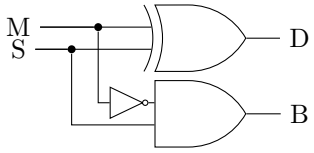


Figure 1.3: One-digit binary half subtractor

Input			Output	
M	S	B_{in}	D	B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	0
1	1	0	0	0
0	0	1	1	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	1

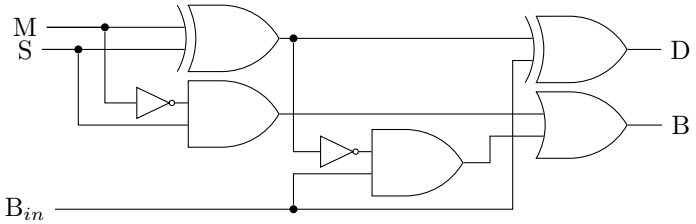


Figure 1.4: One-digit binary subtractor

Subtraction of one from a binary number can be seen as a subtraction with borrow, with the peculiarity that borrowing can also be seen as a bit flip as `1` is the maximum value a digit can have (listing 1.2). In other words, all lower `0`s are turned into `1`s and the lower `1` is turned into `0`, or all bits up to and including the lowest `1` are flipped. If there is no bit set — i.e. the minuend is zero — all bits are turned into `1`s[5]. Digits above the lower `1` are unchanged.

[4]Some architectures take advantage of the arithmetic properties described in section 1.1.2 (*One's and two's complement*) to implement addition and subtraction using the same circuit with just a few extra switches.

[5]The final value of course depends on the number representation of a particular system. In abstract terms, the `1`s extend to infinity. In a real and finite ALU, all bits in the output register are set to `1`, effecting modular arithmetic. The carry flag is also set, allowing the CPU to handle this underflow case in a variety of ways.

```
 0b0111 = 7    0b1100 = 12      0b0010 = 2
-0b0011 = 3   +0b0001 =  1     -0b0100 = 4
-------       -------          -------
 0b0100 = 4    0b1100           0b0010
                   ^^ borrow       ^   borrow
               -------          -------
                0b1011 = 11     0b11110 = 30
                                  ^     borrow flag
                                -------
                                 0b1110 = 30 % 16
                                        = 14
```

Listing 1.2: Subtraction with modular arithmetic

These properties can be taken advantage of to build expressions that perform interesting and useful operations using simple bitwise and arithmetic operations. A bitwise *and* operation can be used to clear the least significant set bit in a number:

```
template<std::unsigned_integral T>
constexpr auto blsr(T n) {
    return n & (n - T{1});
}
```

Listing 1.3: `blsr` using subtraction

```
 0b1100 = 12    0b1100 = 12
-0b0001 =  1   &0b1011 = 11
-------        -------
 0b1011 = 11    0b1000 =  8
```

Listing 1.4: `blsr` operations in detail

The function is named after the instruction in the x86 architecture which performs the same operation. It can in turn be used to check if a number is a power of two — i.e. if it only has one bit set:

```
constexpr bool is_pow2(std::unsigned_integral auto n) {
    return n && !blsr(n);
}
```

Listing 1.5: `is_pow2` using `blsr`

The ability to clear one bit at a time without knowing its position can also be used to reduce the number of operations required for algorithms that operate once per set bit:

```
template<std::unsigned_integral T>
constexpr T popcnt(T n) {
    T ret = 0;
    for(; n; n = blsr(n))
        ++ret;
    return ret;
}
```

Listing 1.6: `popcnt` using `blsr`

The function is similarly named after the corresponding x86 instruction. This implementation is superior — i.e. will perform less steps — to naive ones which iterate over all bits and/or advance one bit at a time:

```
template<std::unsigned_integral T>
constexpr T popcnt(T n) {
    constexpr T w =
        std::numeric_limits<T>::digits;
    T ret = 0;
    // iterate over every bit in `n`
    for(T i = 0; i != w; ++i)
        ret += !!(n & (T{1} << i));
    return ret;
}
```

Listing 1.7: linear popcnt

```
template<std::unsigned_integral T>
constexpr T popcnt(T n) {
    T ret = 0;
    // iterate over the LSBs of
    // `n` up to the leftmost
    // `1` (inclusive)
    for(; n; n >>= 1)
        ret += n & T{1};
    return ret;
}
```

Listing 1.8: popcnt using bitwise shift

Multiplication

The computation of the product of two binary numbers can be separated in several stages (listing 1.9). For a binary number of n digits, the *multiplicand* is first multiplied by each digit of the *multiplier*, generating n *partial products*. As with the carry bit in addition, this is a simple *and* operation, and the result is all zeroes or the multiplicand unchanged, depending on whether the multiplier bit is `0` or `1`.

```
    0b0110 =  6
   *0b0011 =  3
----------
    0b0110 =  6
   0b0110  = 12
  0b0000   =  0
+0b0000    =  0
----------
0b00010010 = 18
```

Listing 1.9: Binary multiplication

The partial products are then multiplied according to the degree of the multiplier digit that generated them. Just as with any base, this is a shift operation for binary numbers. This also means multiplication, at least from this point on, has to be done using integers of twice the width of its inputs. In hardware or software this usually means the destination is either a single element of a larger type or two elements of the same type as the input (e.g. an area of a register with twice the width or two separate registers of the same width).

The final product is the sum of all n partial products. This is analogous to the long multiplication method for decimal numbers, but simpler since all operations on binary numbers can be reduced to logical or shift operations. Simple architectures reuse the shifter and adder to accumulate partial products one-by-one. Faster multipliers have extra circuitry to compute partial products in parallel and sum them together. Figure 1.5 shows a simple circuit that multiplies two 2-bit numbers.

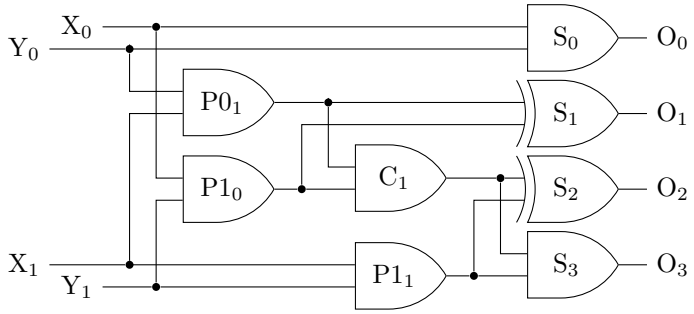


Figure 1.5: 2-digit binary multiplier

Division

Division is the most complex of the simple arithmetic operations, although, as is the case for the other operations, it is considerably simpler in binary. Similarly to multiplication, the division of a *dividend* by a *divisor* follows the long division algorithm also used for decimal numbers (figure 1.10).

First, the divisor is left-aligned with the dividend: a shift operation based on the width of its value. It is then compared to the corresponding most significant bits of the dividend to determine if it is greater than or equal to it. The result of the comparison, zero or one, is the first bit of the quotient. That bit is multiplied by the divisor (which, as we have seen in the multiplication case, is either zero or the divisor unchanged) and subtracted from the portion of the dividend that was used in the comparison.

This process is repeated, with both the quotient and the dividend being shifted up by one position at each iteration, until all bits in the dividend have been processed (i.e. it is right-aligned with the divisor). The remainder is whatever is left in the most significant part of the dividend after the last subtraction.

This process is significantly more complicated than the previous arithmetic operations as it requires iteration and comparisons, has a variable number of steps (depending on the width of the *value* of the dividend/divisor), and each step is dependent on the result of the previous. Still, it is fundamentally composed of the same simple operations used throughout this section.

```
 11000101 | 1010
----------------------
 11000101 | cmp: <
-1010     | q = 1
-----     |
 00100101 | shift
 0100101  | cmp: >
-0000     | q = 10
-----     |
 0100101  | shift
 100101   | cmp: >
-0000     | q = 100
-----     |
 100101   | shift
100101    | cmp: <
-1010     | q = 1001
------    |
 10001    | shift
10001     | cmp: <
-1010     | q = 10011
-----     |
 0111     | r = 111
```

Listing 1.10: Binary division

1.1.2 One's and two's complement

All previous examples focused on natural (i.e. zero or positive) numbers. Negative numbers are defined as *additive inverses*: $-x$ is the number y such that $x + y = 0$. We have observed this property previously in modular arithmetic: $-1 \equiv_{\mathrm{mod}\ 16} 15$, since $-1 + 16 = 15$ and $1 + 15 = 16 \equiv_{\mathrm{mod}\ 16} 0$. Similarly, for every possible value of a 4-bit integer:

...	-16	0	16	...	$\equiv_{\text{mod } 16} 0 \quad = 0000_2$
...	-15	1	17	...	$\equiv_{\text{mod } 16} 1 \quad = 0001_2$
...	-14	2	18	...	$\equiv_{\text{mod } 16} 2 \quad = 0010_2$
...	-13	3	19	...	$\equiv_{\text{mod } 16} 3 \quad = 0011_2$
...	-12	4	20	...	$\equiv_{\text{mod } 16} 4 \quad = 0100_2$
...	-11	5	21	...	$\equiv_{\text{mod } 16} 5 \quad = 0101_2$
...	-10	6	22	...	$\equiv_{\text{mod } 16} 6 \quad = 0110_2$
...	-9	7	23	...	$\equiv_{\text{mod } 16} 7 \quad = 0111_2$
...	-8	8	24	...	$\equiv_{\text{mod } 16} 8 \quad = 1000_2$
...	-7	9	25	...	$\equiv_{\text{mod } 16} 9 \quad = 1001_2$
...	-6	10	26	...	$\equiv_{\text{mod } 16} 10 \quad = 1010_2$
...	-5	11	27	...	$\equiv_{\text{mod } 16} 11 \quad = 1011_2$
...	-4	12	28	...	$\equiv_{\text{mod } 16} 12 \quad = 1100_2$
...	-3	13	29	...	$\equiv_{\text{mod } 16} 13 \quad = 1101_2$
...	-2	14	30	...	$\equiv_{\text{mod } 16} 14 \quad = 1110_2$
...	-1	15	31	...	$\equiv_{\text{mod } 16} 15 \quad = 1111_2$

This naturally leads to partitioning the space of possible values in the middle, obtaining an equal number of negative and zero/positive integers. Values 0...7 (0000_2...0111_2) are positive with the most significant bit equal to zero, while values -8...-1 (1000_2...1111_2) are negative with the most significant bit equal to one. Note that the minimum value is one less than the negated maximum value, since the value zero takes the place of one of the positive values. That is, the minimum value for an n-bit number is -2^{n-1}, while the maximum value is $2^{n-1} - 1$. It can be verified that a number added to its negative results in zero:

$$\begin{aligned} 7 + (-7) &= 0111_2 + 1001_2 \\ &= 10000_2 \\ &= 16 \\ &\equiv_{\text{mod } 16} 0 \end{aligned}$$

From this we also derive how to obtain a negative number from its binary representation: complement all the bits and add one.

$$\begin{aligned} 7 &= 0111_2 \\ &= -(\sim 0111_2 + 1) \\ &= -(1000_2 + 1) \\ &= -(1001_2) \end{aligned}$$

The complement operation, commonly spelled $\sim$ in programming languages, also called the *diminished radix complement*, is the process of obtaining the complement of a digit with a given radix r by subtracting it from $r - 1$. In decimal numbers, $r = 10$ and the *nine's complement* of a digit x is $9 - x$ ($9 - 7 = 2$, $2 + 7 = 9$). In binary, $r = 2$ and the *one's complement* of a digit x is $1 - x$ ($1 - 0 = 1$, $1 - 1 = 0$). This is the same as inverting the value of the bit. In the same manner, for an n-digit number of radix r the diminished radix complement can be obtained by subtracting it from $r^n - 1$ ($\sim 0111_2 = 1111_2 - 0111_2 = 1000_2$).

The process of obtaining the negative number described previously is called the *radix complement*, and consists of subtracting the n-digit number in radix r from r^n. It is equivalent to the diminished radix complement plus one. In binary, $r = 2$ and the *two's complement* of a number x of n digits is $2^n - x$.

One peculiarity of the two's complement representation is that there are *two* numbers whose negation equals itself. One of them is zero, which is expected from regular arithmetic (since $2^n - 0 = 2^n \equiv_{\bmod 2^n} 0$), but also the minimum (negative) integer value.

$$\begin{aligned} 2^n - (-2^{n-1}) &= 2^n + 2^{n-1} \\ &\equiv_{\bmod 2^n} 2^{n-1} \\ &\equiv_{\bmod 2^n} 2^{n-1} - 2^n \\ &\equiv_{\bmod 2^n} -2^{n-1} \end{aligned}$$

Because the sign bit is special, there are two possible ways to implement the bitwise shift right operation. Shifting a negative number left has the same effect as doubling it (including shifting the minimum value, which turns into zero, as $2 * -2^{n-1} = -2^n \equiv_{\bmod 2^n} 0$). But shifting to the right also moves the sign bit, turning a negative number into a large positive one. Some architectures provide the *arithmetic shift* operation, which is similar to the regular shift but propagates the sign bit instead of moving it and filling the left side with zeroes: `x >> n` sets the highest $n + 1$ bits the value of the sign bit.

Yet another possible way to interpret an n-digit binary number x in two's complement is to apply the regular n-digit polynomial to all bits except for the sign bit, which has its value negated.

$$x = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

A shortcut when manually calculating the two's complement of a number is to copy all least significant bits up to and including the rightmost `1`, then invert all digits to the left. This is because, similarly to the reverse process of subtracting one described in section 1.1.1, the one's complement will turn `0`s into `1`s, and adding `1` to form the two's complement will turn all of them back to `0` and carry until the first `0`, which will be turned back into a `1`.

As mentioned in section 1.1.1 (*Binary numbers and modular arithmetic*), the properties of numbers in two's complement can be exploited to produce a unit that functions as both an adder and a subtractor. This is because, as we have seen, the subtraction $X - Y$ in modular arithmetic is the same as the addition $X + (N - Y)$, where N is the number of possible integer values. It can also be written as $X + \sim Y + 1$, where $\sim Y$ is Y's complement as in C, as $(N - 1) - Y$ is also the complement operation. Given these equalities, a circuit can be built with a switch that will complement the second argument and set the carry flag, allowing it to perform additions and subtractions with almost the same logic.

Multiplication requires just a few changes to work with integers in two's complement. Listing 1.11 expands listing 1.9 to show cases where the multiplicand and/or the multiplier are negative. For the former case, the partial products is unchanged, but products must be sign-extended to the full width of the resulting type when they are added. For the latter, the partial product

resulting from the MSB of the multiplier is negated, in accordance with the digit sum formula for two's complement integers presented earlier.

```
    0b0110 =  6        0b0100 =   4        0b1100 =  -4
   *0b0011 =  3       *0b1101 =  -3       *0b1101 =  -3
----------          ----------          ----------
    0b0110 =  6        0b0100 =   4    0b11111100 =  -4
   0b0110  = 12       0b0000  =   0       0b0000  =   0
  0b0000   =  0    + 0b0100   =  16   +0b111100   = -16
+0b0000    =  0    -0b0100    =  32   -0b11100    = -32
----------          ----------          ----------
0b00010010 = 18       0b10100 =  20    0b11101100 = -20
                   -0b0100000 =  32   -0b11100000 = -32
                   ----------          ----------
                   0b11110100 = -12    0b00001100 =  12
```

Listing 1.11: Signed binary multiplication

1.1.3 Exercises

1. The return value of the `blsr` function in listing 1.3 is the same type as its input.

   1.1. Does that provide an acceptable range of values? If not, what type would be most appropriate for the return value? What is the relationship between the type of the input and range of possible returned values?

   1.2. Write a type metafunction (v. section 5.2, *Metaprogramming*) that returns the appropriate type to store the number of bits in an unsigned integer of a given type.

2. Describe possible forms of storage for the state of the board in a game of chess and their advantages.

1.2 Floating-point numbers

The representation of binary floating-point numbers as defined by the IEEE-754 standard is composed of three elements: 1) sign bit, 2) exponent, and 3) mantissa. The sizes and ranges for the second and third values depend on the overall storage size reserved for the number; common sizes are shown in table 1.1.

	Common name	C type	Total bits	Exponent	Mantissa
`f16`	half precision	—	16	5	10
`f32`	single precision	`float`	32	8	23
`f64`	double precision	`double`	64	11	52
`f128`	quadruple precision	`long double`	128	15	112

Table 1.1: Common floating-point sizes

Note that the C standard does not require any of the three types in the table, nor that they be implemented using IEEE-754 (but if they are, those must be the corresponding types). Additionally, if 128-bit numbers are not supported, `long double` may be implemented as an unspecified "extended" format whose only requirement is greater precision than a `double` (e.g. the x87 80-bit extended precision format). Regardless of the size, the value of a binary floating-point number is always calculated in the same manner:

$$f = s\ b^e m$$

In reality, these components are not stored in their original form. The most significant bit records the sign s, just as in one and two's complement representations (v. section 1.1.2, *One's and two's complement*). The base b is always implicitly 2, and the exponent e is stored with a *bias* (v. section 1.2.1, *Exponent*). Only the fractional bits of the mantissa m are stored, an implicit leading digit of 1 is always added. Thus, for a single-precision number (`f32`), the value based on the stored components is:

$$(-1)^s\ 2^{e-127}\left(1 + m\ 2^{-23}\right)$$

Where m is the stored bits of the mantissa interpreted as an unsigned integer. An alternate formulation, where m_i means "the ith bit of the stored mantissa", is:

$$(-1)^s\ 2^{e-127}\left(1 + \sum_{i=1}^{23} m_{23-i}2^{-i}\right)$$

From these equations, the minimum and maximum (absolute) representable values can be derived. The minimum value has an exponent of -126 and a significand of 1 (i.e. the stored mantissa is zero), while the maximum value has an exponent of 127 and a mantissa whose bits are all ones:

$$\begin{aligned}
\texttt{FLT_MIN} &= \texttt{0 00000001 00000000000000000000000}_2 \\
&= \texttt{0080 0000}_{16} \\
&= 2^{-126} \\
&\approx 1.1754943508223 \times 10^{-38} \\
\texttt{FLT_MAX} &= \texttt{0 11111110 11111111111111111111111}_2 \\
&= \texttt{7f7f ffff}_{16} \\
&= 2^{128} - 2^{128-24} \\
&\approx 3.4028234663853 \times 10^{38}
\end{aligned}$$

The following C code can be used to extract each of the components above from a single-precision value and reassemble them into the original number. In the second part, size limits are disregarded in the calculation of the exponent and significand for simplicity of demonstration. The standard library provides several functions in `<math.h>` which perform this type of operations; they are shown in comments next to their equivalent.

```
extern float f;
u32 uf;
```

```
static_assert(sizeof(uf) == sizeof(f));
memcpy(&uf, &f, sizeof(uf));
const i32 s = uf >> 31; // (bool)signbit(f)
const i32 e = ((uf >> 23) & 0xff) - 127; // ilogbf(f)
const i32 m = uf & 0x7fffff;


const float fs = 1 - (s << 1); // copysignf(1, f)
const float fe = 1 << e; // scalbnf(1, e)
const float fm = 1 + m / 0x1p23f; // 1 + scalbnf(m, -23)
f = fs * fe * fm;
```

1.2.1 Exponent

The representation of the exponent, which can be a negative number, is different from the ones shown in section 1.1.2 (*One's and two's complement*): the stored value is instead *biased*. For single-precision numbers, with 8-bit exponents, the actual stored value is the exponent plus 127. Special treatment is given to numbers with an encoded exponent value of zero (v. section 1.2.3, *Subnormal numbers*) and 255 (the maximum value, used for infinity and `NaN` values), so the effective exponent range is $[1, 254] - 127 = [-126, 127]$. This range has one more value in the non-negative range than the negative range, also unlike two's complement.

The main advantage of this storage format is that numbers in the normal range (i.e. excluding `NaN`s and infinities) can be compared lexicographically directly in their encoded format as if they were regular signed integers. This would not be possible in a complement representation since the (exponent's) sign bit would place negative numbers after non-negative numbers.

1.2.2 Precision

As any numerical representation in computers, floating-point numbers can represent only a finite set of values. However, because of the exponent multiplication and the fixed number of bits in the mantissa, the range of this precision is variable. This is manifested in two ways. First, if a large portion of the bits of the mantissa are required to represent the integral part, few bits will be available for the fractional part, resulting in less precision for fractional digits. The absolute precision range also depends on the magnitude of the number, since the number of bits in the mantissa is constant: the larger the number, the larger the exponentiated multiplier, and the larger the gap between each value representable by the mantissa.

A visual conceptualization of the IEEE-754 storage format can be helpful in understanding these precision limitations. The significand has an implicit digit of one, while the mantissa represents fractional digits, meaning the effective range of the significand is [1, 2). A crucial observation is that, for any number whose exponent is e and whose mantissa is all ones, the next representable value has exponent $e + 1$ and a mantissa of zero[6]. This means every value which is representable also has a *unique* representation.

In this perspective, each exponent value can be seen as a sub-range of the entire range of representable values (i.e. [`FLT_MIN`,`FLT_MAX`]). Each covers the range $[2^e, 2^{e+1})$, where e is the value of the exponent, and is in turn divided equally into 2^n sub-ranges, where n is the number of bits in the mantissa[7]. It is thus obvious (hopefully) why the precision is variable according

[6] In fact, the bit representation of the number can simply be treated as an unsigned integer in this operation, which is equivalent to the standard function `nextafter`, for all values in the range [`0`, `FLT_MAX`]. Similarly, the same is true in the other direction for negative numbers in the range [`-FLT_MAX`, `-0`], although note that there is no such relation in the transition between negative and positive values in either direction. See figure 1.6 (*Adjacent floating-point ranges*).

to the magnitude of the number: each sub-range has the same number of divisions, so larger sub-ranges will have larger intervals between each division.

As an extreme example, numbers close to `FLT_MAX` have, as shown above, an exponent of 127. The smallest number with that exponent is 2^{127}, which in unsigned integer notation is `0x80000000000000000000000000000000`, i.e. a one followed by 127 zeroes. In other words, the leading one bit implicit in the significand is shifted left 127 positions. The next representable number is the one where the mantissa has its least significant bit set, i.e. $2^{127}\ (1 + 2^{-23})$. The interval between those numbers is $2^{127}\ 2^{-23} = 2^{127-23} = 2^{104}$. If these numbers are involved in additions/subtractions with others which contain any extra digits inside this enormous interval, those digits will be discarded since there is not enough precision in the mantissa to represent them. In general, operations between numbers of similar magnitude have the most precision.

1.2.3 Subnormal numbers

In order to cover the range $(-\texttt{FLT\_MIN}, \texttt{FLT\_MIN})$, IEEE-754 includes as an extension a category of numbers called *subnormal*. These numbers have a biased exponent of zero, but their effective exponent is considered to be one and the implicit leading bit in the significand is not added. We can now depict the full range of values representable by single-precision IEE-754 numbers (figure 1.6).

1.2.4 Machine epsilon

A standard measurement of the precision of floating-point representations is the *machine epsilon*: the distance between the value 1 and the next smallest value that is greater than it, i.e. the smallest value ϵ which satisfies $1 + \epsilon \neq 1$. For an IEEE-754 single-precision number, the previous section has shown that $\epsilon = 2^{-23}$.

Because the precision of a floating-point number varies according to its magnitude, this distance increases as a function of the exponent. In general, the absolute distance between a floating-point number with exponent e and the next representable value is $2^e\epsilon$. This value can be used to calculate relative error, turn greater-than-or-equal comparisons into greater-than, etc.

1.3 Latches

All logic circuits discussed so far have been directed acyclic graphs: data flow in a single direction for each connection and any individual signal starts at one of the inputs and travels through the graph to one of the outputs, never passing through the same gate more than once. This is enough to implement transient computation, such as additions and bitwise operations, as we have seen, but computers also need to *retain* information for a period of time. Building components which can do that requires some form of loop where the current state is used in the computation of the next.

A very simple circuit with that property is an *or* gate whose output is connected to one of its inputs (figure 1.7). The output of this circuit will be zero until the input is set. From that point on, as long as the circuit is powered, its output will always be on, since the signal that loops back to the *or* gate is on and that is enough to turn the gate on.

Figure 1.8 shows an improved version of this concept where two *nor* gates are interconnected. Initially, one of the two outputs, which are complements, is set (which one depends on propagation delays, this is a hardware race condition).

[7] The value of the mantissa can also be considered an *offset* into the range denoted by the exponent, or

Assuming that is the gate associated with R, the circuit will reach a stable state where Q is set. Raising and lowering S at this point has no effect, as the *nor* gate will not propagate the signal as long as the crossed connection is on. Raising R (short for *reset*) will clear Q, but also clear the input to the other *nor* gate. This will cause both $\bar{Q}$ and one of the inputs to the first *nor* gate to be set. R can be lowered and raising and lowering it again at this point has no effect. Raising S (short for *set*) has the opposite effect: $\bar{Q}$ and the first gate are cleared, Q is set, and we are back at the initial state.

From this we can observe that the circuit has two stable states: Q set and $\bar{Q}$ cleared, or vice versa. Raising R or S transitions the circuit from one state to the other, which will be maintained until the next transition. Components that exhibit this type of behavior are called *latches*, as the output is locked when an operation is performed and state is preserved. This particular circuit is called an *SR latch*, due to its two inputs.

The *D latch* (figure 1.9) is a similar circuit which uses a single input instead of a reset/set pair. Two additions to the SR latch accomplish this. First, a single input D (short for *data*) is connected to R and S, with R going through a *not* gate. State is now no longer retained since toggling D immediately raises one of R or S. An input EN (short for *enable*) is added so that transitions occur only when it is on. This is done by placing two *and* gates between R/S and the latch, which are also connected to EN.

These additions create a circuit that will change its state based on the combination of both its inputs. Figure 1.10 shows the relationship between changes to D/EN and the output of a D latch over time. Whenever EN is on, Q follows the value of D (green lines). Conversely, changing D while EN is off has no effect on the value of Q (red lines). This type of latch is called a *transparent latch*, as the value of the input is constantly forwarded to the output as long as the enable signal is on.

A more common design in computers is to have components synchronize state changes according to a regular, intermittent signal: a *clock*. In this scenario, we want latches to update their state at the *edge* of the clock signal, i.e. right when it changes its state — either from zero to one, the *rising edge*, or from one to zero, the *falling edge*. Figure 1.11 shows a timing diagram with the same input, but with Q transitions happening at the clock rise signal. The intervals in which D influences the value of Q (green lines again) are now very short and even more D transitions are ignored.

A simple circuit that can identify the rising edge of a clock signal, an *edge detection circuit*, is shown in figure 1.12. At first it might seem like a circuit that never outputs anything, but the small delay introduced by the *not* gate allows the output to be turned on momentarily. More *not* gates can be added to increase the interval in which the input signal is allowed to be propagated. Other types of edge detection circuits are possible, such as placing a capacitor and a resistor between the input and output. Substituting the EN input of a latch with such an edge detector creates a component called a *flip-flop*, which otherwise operates in the same way: a transparent latch is *level-triggered*, while a flip-flop is *edge-triggered*. These are the basic building blocks for storing data in a logic circuit.

1.4 Memory

1.4.1 Virtual memory

Modern operating systems for most machines which are not small embedded platforms do not expose memory directly to processes. Instead, a hardware component called a *Memory Man-*

conversely, the exponent can also be seen as *shifting* the significant digits of the mantissa.

agement Unit (MMU) mediates access to physical memory. Memory as seen by the processes is called *virtual memory*. The MMU translates the *virtual memory addresses* used by the processes into real, *physical memory addresses* according to the configuration done by the operating system. Virtual memory is thus a software-controlled set of memory addresses.

This mapping is not restricted to simply address translation. Any time an access to an unmapped address occurs, a *page fault* is generated (memory is divided in pages, as will be explained later). This is analogous to a hardware interrupt, and the operating system can configure a handler which will direct the MMU to take one of several possible actions to resolve the address:

- Immediately assign a physical memory address and service the memory request. This is called a *soft page fault*.
- Defer the decision to the operating system. This is called a *hard page fault*.
- Immediately fail the memory access. This is the source of the (in)famous *segmentation fault* or *access violation* failure.

This level of indirection is only practically possible because it is implemented in part in hardware. It is nonetheless immensely useful: the possibility of deferral of faults to a software process allows the implementation of several mechanisms which are widely used in modern operating systems and programs.

Virtual address space

From the operating system's perspective, one of the most important is the possibility of setting up distinct mappings for each process, which are also separate from that of the kernel. This is the fundamental method of separation between processes and between user and kernel space. Because of the unprivileged nature of user mode (MMU configuration is only allowed when the CPU is in kernel mode), the operating system can reconfigure mappings whenever it schedules a new process to guarantee it only has access to its own view of memory. Kernel memory is thus protected from user space access, and each process is restricted to its own address space[8].

This level of indirection in memory accesses can also be used to radically change the contents of physical memory in a way that is transparent to the user processes:

- Since there is no relation between virtual and physical addresses, a large contiguous memory allocation can be serviced by dividing it into several smaller ones, potentially reducing overall memory fragmentation.
- Process data do not even have to reside in memory at all: when system memory is scarce — either due to allocations or caching — less-frequently-used memory regions can be move to slower storage (such as a hard drive) to make space and if/when they are requested again, a process called *swapping*.

  Doing so takes advantage of *locality of reference*, a common memory access pattern in the execution of programs: memory tends to be referenced locally repeatedly, either spatially (adjacent regions) or temporally (repeating accesses, such as in a loop). Only the relevant, relatively small regions which are presently referenced need to be in memory, in what is called the *resident set* of the process[9].

[8] Recently, even this coarse dichotomy has been deemed insufficient, and further separation between distinct areas of kernel memory is being considered, e.g. `https://lwn.net/Articles/886494/`.

[9] As displayed, for example, in the `rss` (resident set size) field in `ps(1)` or the `RES` column in `top(1)`.

- The operating system and the MMU may also support *memory protection*, usually in the form of several permission bits which control access to certain memory regions. Protection bits can also be used to make regions read-only, prevent their execution as code, completely forbid access, etc.

- Regions can be shared by multiple processes. This can happen when multiple processes are created for the same program or use the same shared library (the text segment can be shared), as a result of the `fork` system call (all memory is initially shared), or via explicit requests (using `shmget(2)`, `mmap(2)`, etc.). These common regions can be set up in either *private* or *shared* mode. Modifications made to private regions are not visible to other processes: the region is duplicated by the operating system when it is first written.

- Memory regions do not have to be allocated or initialized immediately: they can simply be reserved and serviced when needed. For example, a special segment in a process' address space is the BSS segment (the name originates historically from *block started by symbol*), a region of memory which is guaranteed to be zero-initialized[10]. When a process is started, this region is not allocated by the operating system: it is simply marked as "zero-initialized" and materialized on first access.

- Files can be directly mapped as part of the virtual address space of a process. This can reduce the overhead of making multiple system calls to read and write data: the access can be made as if the contents of the file were a regular memory region and is handled transparently by the MMU.

Pages

Due to the size of physical memory commonly found in current systems — on the order of billions of bytes (GiB) — it is prohibitive to treat each memory address individually. Memory is instead divided into *pages*: contiguous regions of a predefined size. A common page size, used in the Linux kernel, is 4096 bytes (4KiB)[11]. This affects several aspects of the system: many operations are required to be *page-aligned* — i.e. are restricted to begin at page boundaries — for this reason. Similarly, MMU configuration is usually done in terms of pages, in what are called *page tables*.

Even that, however, is not sufficient. In a 32-bit system — where a space of 4GiB is addressable — with 4KiB pages, 2^{20} entries would be necessary ($2^{32}/2^{12} = 2^{20}$). That overhead would be prohibitive for any non-trivial amount of per-page information, and hopeless in a 64-bit address space. Configuration is done instead in multiple levels: virtual addresses are split in several parts, called *page directories*, one for each level. Directories can be sparsely allocated, and a single entry can be used for several contiguous pages, making it possible for mappings of several processes to remain in memory[12]. These directories are usually walked in hardware by the MMU to resolve virtual addresses.

TLB

Even though virtual memory accesses are mostly performed by hardware components, doing a full translation on each access would impose a significant overhead. For this reason, the CPU

[10] In C, it corresponds to variables of static storage duration declared `extern` and default- or zero-initialized.

[11] Linux also has *huge pages*, which are considerably larger (anywhere from 2MiB to 1GiB), to mitigate some of the problems described in this section.

[12] Although note that *address space layout randomization* (ASLR), a security measure present in modern operating systems, conflicts with this goal.

itself employs a cache for some of the page table entries (e.g. those in its L1 cache). This cache is called the *translation lookaside buffer* (TLB). Memory accesses whose address are found in the cache go directly to main memory (or to one of the memory caches). A cache miss triggers a page lookup as described above; this is an expensive process and frequent TLB misses can significantly affect the performance of a program. This is one of the reasons for the inefficiency of context switches and process scheduling: they can result in a partial or complete TLB flush, since a process cannot reuse the virtual memory mappings of another one.

1.4.2 Cache

Memory caches are a consequence of the frequency discrepancy between CPU and memory units. Initially, these operated at comparable speeds, but as a consequence of Moore's law processor speeds increased dramatically, greatly surpassing the improvements in memory units. A secondary effect was that CPU frequencies increased such that the distance between these components became a limiting factor. Even imagining a scenario where transmission occurred at the speed of light and in a vacuum — an already unrealistic scenario — the transmission time for a signal between two components at a $1cm$ distance would be:

$$c \approx 3 \times 10^8 m/s$$
$$F = \frac{1cm}{c} \approx \frac{10^{-2}m}{3 \times 10^8 m/s} \approx 3 \times 10^{10} Hz$$

That is, even in ideal conditions a $1cm$ distance limits communication to the low gigahertz (GHz) range, which is close to the frequency common in modern processors. These two factors combined result in CPUs which are orders of magnitude (usually two or more) faster than the circuits that connect them to each other and memory systems. To avoid these long interruptions in execution whenever a memory access occurs, modern processors have multiple levels of *cache*.

Two main characteristics distinguish memory caches from main memory: their type and their size. System memory is almost exclusively *dynamic RAM* (DRAM), making large storage arrays cheap and compact, but slow. Caches, on the other hand, operate on smaller sets of data at a time, so have lower storage requirements. They are usually built with *static RAM* (SRAM), which are much more complex, but much faster. Their reduced size (even with the extra components compared to DRAM) also allows them to be placed closer to the CPUs, reducing transmission times. CPUs typically have multiple levels of cache (`L1`, `L2`, etc.), each larger but further, increasing both storage capacity and access times. Higher levels can also be shared between two or more processors. Common access times for the L1 and L2 caches are around one and ten cycles, respectively.

In addition, it is common for two separate types of cache to be used: *instruction* and *data* caches (sometimes called *icache* and *dcache*). Program text and data have distinct access patterns and often reside in different regions of memory, so this separation can result in better cache utilization. The instruction cache can be used to store decoded instructions, speeding up the execution, since instruction decoding is relatively slow, and reducing latency, especially in case of a pipeline stall.

Cache systems with multiple layers can be either *inclusive* or *exclusive*, depending on whether data present in lower (i.e. smaller) layers are also present in upper layers. Inclusive caches propagate data across all layers, which exclusive caches can have data stored only in some layers. Evictions in inclusive caches are faster, since data can simply be flushed to the next layer. Conversely, loading data is faster in exclusive caches, since only a single layer is affected.

Lines

Internally, caches operate on fixed-length blocks called *cache lines*. The length varies by architecture, but is almost always a power of 2 to facilitate operations (16 to 256 byte cache lines are common, 64 being the most common). Whenever there is an access to a memory location, the address is analyzed and matched to the entries in the cache. The first access will not find the line in the cache (a cache *miss*), resulting in a stall as the line is transferred from main memory (or upper layers of cache). Subsequent accesses will use the contents of the line in the cache (a *hit*), as long as it is not evicted.

The mapping of memory addresses to positions in the cache is determined by the cache *associativity*. The memory address is partitioned into different pieces depending on the various sizes of the components involved. On a Linux machine, these sizes can be determined via `sysfs`:

```
$ d=/sys/devices/system/cpu/cpu0/cache/index0
$ cat $d/coherency_line_size
64
$ cat $d/size
32K
```

In a *fully-associative* cache, each memory address can occupy any position in the cache. This allows maximum utilization and hit rate, but it also means a full search through the cache has to be performed to find a given line. These searches are often done by a circuit that tests all values in parallel, so complexity and power consumption make this arrangement only practical for very small caches.

In contrast, in a *direct-mapped* cache, each memory address has a single corresponding position in the cache. In the example above, the 32KB cache can hold 512 64-byte cache lines ($32\text{KB}/64 = 2^{15}/2^6 = 2^9 = 512$). For a given memory address, the lower 6 bits ($2^6 = 64$) are an offset into the cache line and are not used in the cache translation. The next 9 bits ($2^9 = 512$) indicate the position of the line in the cache, called the cache *set*. The remaining bits (17 or 49 for 32- and 64-bit addresses) are the *tag* used to identify to which memory address a given line in the cache corresponds. A direct-mapped cache is analogous to a hash table with no conflict resolution: if the same "hash" position (i.e. set) is already occupied, it is simply evicted and replaced with the new one.

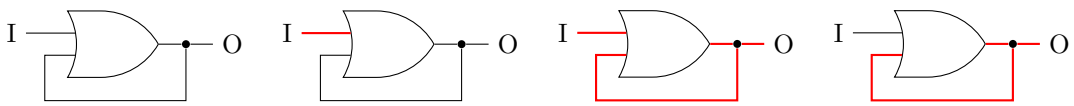
A *set-associative* is a compromise between the simplicity of a direct-mapped cache and the benefits of a fully-associative cache. Each set contains multiple cache lines: after the set is determined for a given address, it can be placed in any of the lines in the set. It is analogous to a hash table with fixed-size buckets and no external chaining. A direct-mapped cache is a 1-way set-associative cache; a fully-associative cache with n sets is an n-way set-associative cache. The cache in the example above in reality is 8-way associative, which means each set has space for eight lines and thus it has not 512, but 64 sets ($32\text{KB}/64/8 = 2^{15}/2^6/2^3 = 2^6 = 64$).

```
$ cat $d/ways_of_associativity
8
$ cat $d/number_of_sets
64
```

The partitioning of the address follows the same rules as before, only now with the reduced number of sets: 6 bits for the line offset (unchanged), 6 bits for the set, and 20/52 bits for the tag.

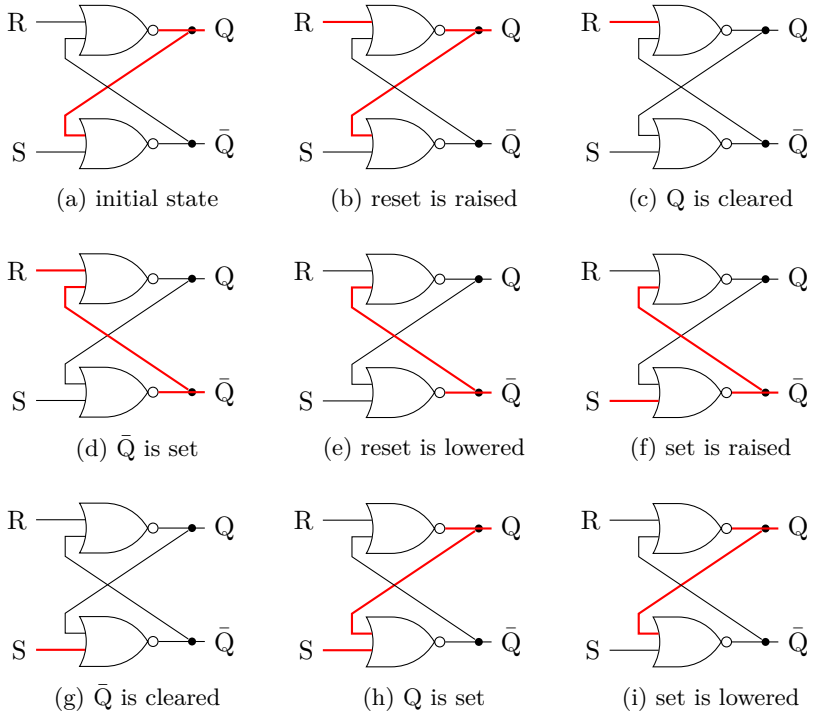
$$
\begin{aligned}
&1\ 11111111\ 00000000000000000000000_2 = -\texttt{INFINITY}\\
&1\ 11111110\ 11111111111111111111111_2 = -\texttt{FLT\_MAX} = -2^{128} + 2^{128-24}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdots\\
&1\ 01111111\ 00000000000000000000001_2 = -1 - 2^{-23}\\
&1\ 01111111\ 00000000000000000000000_2 = -1\\
&1\ 01111110\ 11111111111111111111111_2 = -1 + 2^{-24}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdots\\
&1\ 00000001\ 00000000000000000000000_2 = -\texttt{FLT\_MIN} = -2^{-126}\\
&1\ 00000000\ 11111111111111111111111_2 = \text{min. subnormal} = -2^{-126}\ (1 + 2^{-23})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdots\\
&1\ 00000000\ 00000000000000000000001_2 = -\texttt{FLT\_TRUE\_MIN} = -2^{-127}\ 2^{-23} = -2^{-140}\\
&1\ 00000000\ 00000000000000000000000_2 = -0\\
&0\ 00000000\ 00000000000000000000000_2 = 0\\
&0\ 00000000\ 00000000000000000000001_2 = \texttt{FLT\_TRUE\_MIN} = 2^{-127}\ 2^{-23} = 2^{-140}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdots\\
&0\ 00000000\ 11111111111111111111111_2 = \text{max. subnormal} = 2^{-126}\ (1 - 2^{-23})\\
&0\ 00000001\ 00000000000000000000000_2 = \texttt{FLT\_MIN} = 2^{-126}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdots\\
&0\ 01111110\ 11111111111111111111111_2 = 1 - 2^{-24}\\
&0\ 01111111\ 00000000000000000000000_2 = 1\\
&0\ 01111111\ 00000000000000000000001_2 = 1 + 2^{-23}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdots\\
&0\ 11111110\ 11111111111111111111111_2 = \texttt{FLT\_MAX} = 2^{128} - 2^{128-24}\\
&0\ 11111111\ 00000000000000000000000_2 = \texttt{INFINITY}
\end{aligned}
$$

Figure 1.6: Adjacent floating-point ranges



(a) initial state (b) input is set (c) *or* propagates (d) input is cleared

Figure 1.7: *or* gate with a loop



(a) initial state (b) reset is raised (c) Q is cleared

(d) $\bar{Q}$ is set (e) reset is lowered (f) set is raised

(g) $\bar{Q}$ is cleared (h) Q is set (i) set is lowered

Figure 1.8: SR latch

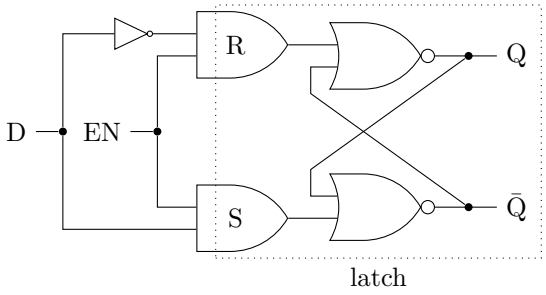


Figure 1.9: D latch

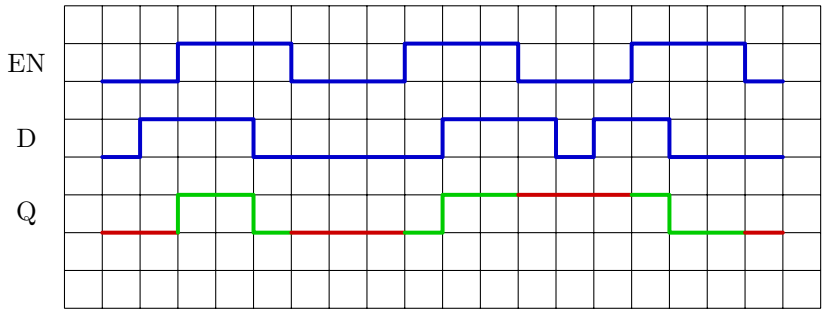


Figure 1.10: D latch timing diagram

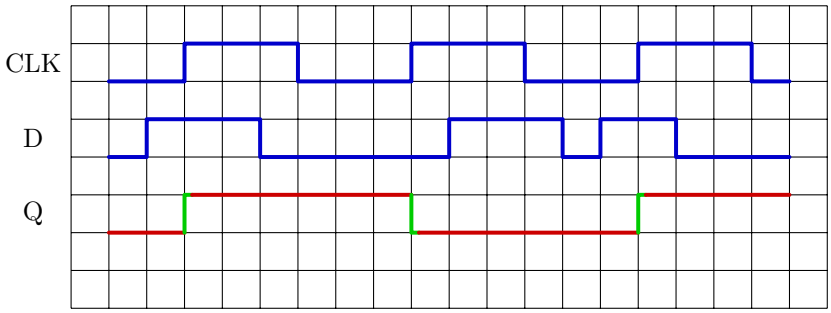


Figure 1.11: Flip-flop timing diagram

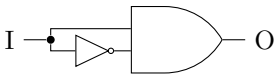


Figure 1.12: *and*/*not* gate edge detector

Chapter 2

Algorithms

2.1 Fundamentals

2.1.1 Intervals and ranges

Both in mathematics and computer science, *ranges* (or *intervals*) of elements are an important concept. The interval corresponding to a sequence of integers such as the set $\{1, 2, 3, 4\}$ can be represented using a few different notations:

0. $[1, 4]$, or $\{x|1 \leq x \leq 4\}$, the integers from 1 to 4, both inclusive
1. $[1, 5)$, or $\{x|1 \leq x < 5\}$, the integers from 1 (inclusive) to 5 (exclusive)
2. $(0, 4]$, or $\{x|0 < x \leq 4\}$, the integers from 0 (exclusive) to 4 (inclusive)
3. $(0, 5)$, or $\{x|0 < x < 5\}$, the integers from 0 to 5, both exclusive

These intervals can be classified into the following categories:

Closed intervals have determinate endpoints (notation 0, whose endpoints are 1 and 4).

Open intervals have no determinate endpoints, only bounds which can be infinitely approached (notation 3, if applied to the set of real numbers, $\mathbb{R}$).

Half-open intervals have both open and closed endpoints (notations 1 and 2).

Left-closed intervals have a minimum element (notations 0 and 1).

Left-open intervals do not (notations 2 and 3).

Right-closed intervals have a maximum element (notations 0 and 2).

Right-open intervals do not (notations 1 and 3).

All of the notation examples above are different ways of representing the same set of integers. One of them, however, is generally preferred to denote ranges in computer programs. The fundamental reason for this is the same that underlies the usage of zero-based indices in most programming languages[1].

[1] Lua being a notable exception where the choice was made deliberately to make the language more approachable to non-technical users (the intended primary audience), ensuring an eternal stream of bugs caused by programmers switching between it and other languages.

Dijkstra 1982 presents several justifications. The set of natural numbers, $\mathbb{N}$[2], has a minimum element (regardless of whether it is assumed to include the number zero or not). Left-open notations need to include an element outside of the set to describe it. Similarly, right-closed notations need to include an element outside of the set to describe an empty interval when the left endpoint is the minimum natural element (since, for example, $[0, 0]$ is the unit set $\{0\}$). This makes the half-open, left-closed notation the preferred one for specifying ranges of values. Beyond these characteristics, it has several other notational advantages (some of which are also true for the other notations in certain cases):

- the length of the interval $[x, y)$ is exactly $y - x$
  - $[x, x)$ is an empty interval
  - an interval of length N can be represented as $[0, N)$
  - an interval of length N starting at x can be represented as $[x, x + N)$
  - $x \bmod N \in [0, N)$
- two intervals $[x, y)$ and $[z, w)$ are adjacent if $y = z$ or $w = x$
  - the concatenation of intervals $[x, y)$ and $[y, z)$ is $[x, z)$
  - an interval $[0, N)$ split at the n-th element results in the intervals $[0, n)$ and $[n, N)$
  - an interval $[x, y)$ split at the n-th element results in the intervals $[x, x+n)$ and $[x+n, y)$
  - an interval $[x, y)$ which can be divided evenly into sub-intervals of size n is the concatenation of the intervals $[x, x + n), [x + n, x + 2n), [x + 2n, x + 3n), \ldots$

When applied to ranges of values in a computer program (e.g. sequences of memory addresses), these properties eliminate most classes of the pernicious *off-by-one* errors. For example, the trivial splitting property makes partitioning a range (in the middle, say, for a binary search) much easier and less prone to errors:

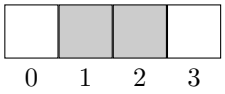
```
// Recursively partition the range [b,e) in half.
void f(int *b, int *e) {
    if(b == e)
        return;
    int *m = b + (e - b) / 2;
    use(m);
    f(b, m), f(m + 1, e);
}
```

Its other beneficial property results in a notation that may seem at first unusual. Consider the depiction of the array in figure 2.1a, and suppose we wish to represent the shaded interval, which consists of the elements at position 1 and 2. Both $[1, 2]$ and $[1, 3)$ seem like equally valid options, i.e. it is not clear whether the end of the range is (or should be) inclusive or exclusive. Enumerating the positions *between* the elements, as in figure 2.1b, results in an unambiguous representation: the interval $[1, 3)$ denotes the range between indices 1 and 3.

Consider also how the end position in this type of half-open range mirrors the form of the traditional `for` loop in a language such as C for not only numbers and addresses, but any type in general:

```
for(int i = b; i < e; ++i)
for(int *p = b; p < e; ++p)
for(T x = b; x < e; ++x)
```

[2]Or, alternatively — and anticipating the conclusion, for programmers: an `unsigned` integer type.



(a) Elements indicated by position

(b) Indices between elements

This is the basis of the rule in the C standard that allows a program to form (but not dereference) pointers to *one past the last element* of an array, and it illustrates the rule more clearly. More importantly for languages such as C++, it is also equivalent to and enables the more general form:

```
for(int i = 0; i != N; ++i)
for(int *p = b; p != e; ++p)
for(T x = b; x != e; ++x)
```

This transformation — which could not happen if $[b, e)$ were a closed interval — of `<` into `!=` may seem inconsequential, but is subtly profound. This is because it no longer limits this construct to contiguous ranges[3]: it can be applied to any range which supports a *sentinel* value as the endpoint, such as a tree or linked list. This is the foundation of iterators and ranges in C++, and the standard library algorithms built on top of them.

Even in less expressive languages, half-open ranges eliminate ambiguities. An insertion at index `i` positions the value between the elements indicated by the index. Inserting at position `0` means prepending, inserting at position `N` means appending. Denoting sub-ranges, e.g. for deletion, is also trivial as shown above.

2.1.2 Utilities

Throughout this chapter, a number of utility functions will be used in the implementation of algorithms. Some will be described as part of each section, but a few basic ones are presented here.

`check_range`

Verifies that an iterator/sentinel pair represents a valid range. It is meant as a debugging aid and only used in assertions. The specific test depends on the types of the inputs:

- Contiguous and random-access iterators are tested using `std::distance`, a constant-time operation for those types.
- A forward iterator needs to be advanced until the sentinel is reached, also done using `std::distance`. This is an invalid operation for invalid ranges, but so will be their use in the algorithms where `check_range` is used. For this reason, the actual value of the distance does not matter.
- Input iterators cannot be checked, since they only yield their value once.

These cases can be nicely discriminated using constraints and concepts from the standard library.

[3] V. the `concepts` in the `std::ranges` namespace of the C++ standard library, such as `forward_range`, `random_access_range`, and `contiguous_range`.

```
template<std::random_access_iterator I, std::sentinel_for<I> S>
constexpr bool check_range(I b, S e) {
    return 0 <= std::distance(b, e);
}

template<std::input_iterator I, std::sentinel_for<I> S>
constexpr bool check_range(I b, S e) {
    if constexpr(std::forward_iterator<I>)
        std::distance(b, e);
    return true;
}
```

contains

Uses `check_range` to determine whether an iterator points to a valid element of a range.

```
template<std::input_iterator I, std::sentinel_for<I> S>
constexpr bool contains(I b, S e, I i) {
    return check_range(b, i) && check_range(i, e);
}
```

is_min_element

Verifies that a given value is the minimum element of a range of values. i.e. that no other element is less than it. It is equivalent to

```
!(std::ranges::min(std::ranges::subrange(b, e)) < x)
```

except it stops at the first element for which the assertion does not hold.

```
template<std::input_iterator I, std::sentinel_for<I> S>
constexpr bool is_min_element(I b, S e, const auto &x) {
    return std::all_of(b, e, [&x](const auto &y) { return !(y < x); });
}
```

min_element

A reimplementation of `std::min_element` for illustrative purposes.

```
template<std::forward_iterator I, std::sentinel_for<I> S>
constexpr I min_element(I b, S e) {
    assert(check_range(b, e));
    if(b == e)
        return b;
    auto ret = b;
    for(auto i = std::next(ret); i != e; ++i) {
        assert(is_min_element(b, i, *ret));
        if(*i < *ret)
            ret = i;
    }
    assert(contains(b, e, ret));
    assert(is_min_element(b, e, *ret));
    return ret;
}
```

lsearch

Used in contrast to bsearch in examples in C.

```
bool lsearch(const int *b, const int *e, int x) {
    for(; b != e; ++b)
        if(*b == x)
            return true;
    return false;
}
```

2.2 Complexity

The single most important graph in algorithmic complexity analysis is shown in figure 2.2. The values of different functions are shown for increasing positive values of the variable n. These functions are used to describe the relationship between an algorithm's resource consumption (for any give resource of interest: number of operations, storage, etc.) and the size of its input. The formal definition of algorithmic complexity uses *Big-O* notation — the complexity of an algorithm is also called its *order*, which has the form $f \in O(g)$. It describes the relation between a function the complexity function f of an algorithm and a function g according to the following principles:

- The notation is asymptotic: $f \in O(g)$ implies that g is an upper bound for f: $f(n) <= g(n)$ for some range of values.
- Input sizes are always positive real numbers, so $0 <= n$.
- The analysis is primarily focused on large input sizes. Some orders may be faster for smaller sizes, but those are rarely the important parts of a program. Therefore, $f \in O(g)$ if there is a value n_0 for which $f(n) <= g(n)$ for all $n > n_0$, that is, even if $f(n) > g(n)$ for some values of n, provided there is a point after which g is always an upper bound for f.
- Similarly, differences between machines are critical factors but very difficult to analyze in the abstract, so $f \in O(g)$ if there is a constant $c > 0$ for which $cf(x) <= g(x)$. That is, even if some constant factor influences the complexity of a specific implementation of an algorithm, it will become irrelevant for large input sizes and can be ignored.
- Usually, only the tightest bound for a function is considered. E.g.: even though it is true that $f \in O(\log n)$ implies $f \in O(n)$ (but not $f \in O(1)$), $O(n)$ usually means $f(n)$ is the most restrictive of the categories that fits the constraints above. In general, for any polynomial, only its highest power of n is considered, as it eventually dominates the value of the function.

Constant complexity, $f(n) = 1$, represents any function that does not depend on the magnitude of its input. A hash table lookup is an example of this complexity: if the load factor of the table is managed, the number of operations performed is independent of the number of elements in the table. Even though this function is usually represented by

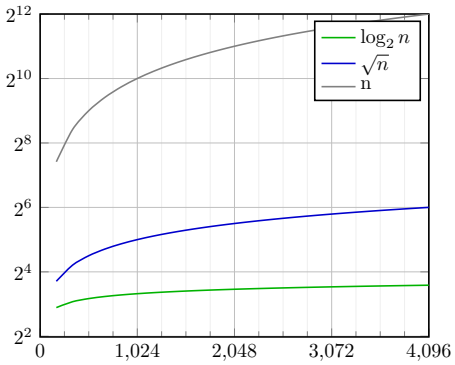


Figure 2.3: $n \log_2 n$ vs. $\sqrt{n}$

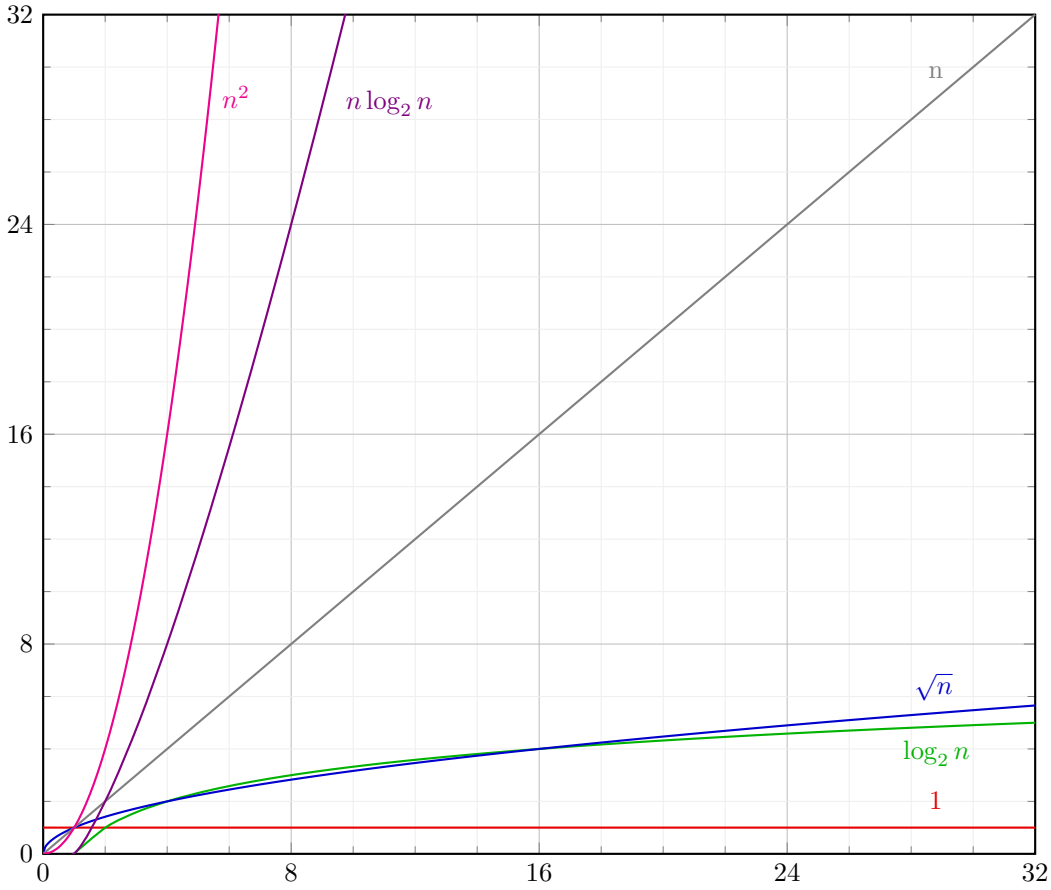


Figure 2.2: Algorithmic complexity functions

the constant 1, from the rules above we derive that any constant function is in the same category.

Linear complexity, $f(n) = n$, is a function directly proportional to its input. Algorithms commonly operate on every element, consume one unit of space, or otherwise perform an operation for each of their input values, making this the most common function to describe their complexity, and a reference to which other functions are compared. A linear search over an array of elements is an example of this complexity.

Different interesting subsets of these functions can be analyzed in separation so that the scale of the graphs can be adjusted, as they quickly become wildly divergent outside the range displayed in the graph. Figure 2.3 compares the lower part of the original graph to the growth of n, this time using a logarithmic scale.

Logarithmic complexity (referring as always in this section to the logarithm base 2 unless given an explicit base), $f(n) = \log_2 n$, is the pattern of algorithms that reduce their domain by some constant factor in each iteration, usually by half. A binary search is an example of this complexity.

This is a very important function that is common of optimized algorithms which dramatically reduces resource consumption, as the $\log_2$ remains a relatively low number, very similar to a constant factor, even for very large (in terms of numbers that computers usually deal with) values of n. As an example, the two most common word sizes currently are 32 and 64 bits. These can address arrays of roughly two million and ten quintillion bytes, but the logarithm base two of those values is, obviously, 32 and 64. This demonstrates both that doubling the logarithm has an enormous effect on the corresponding value of n and that the logarithm is relatively small for very large values of n.

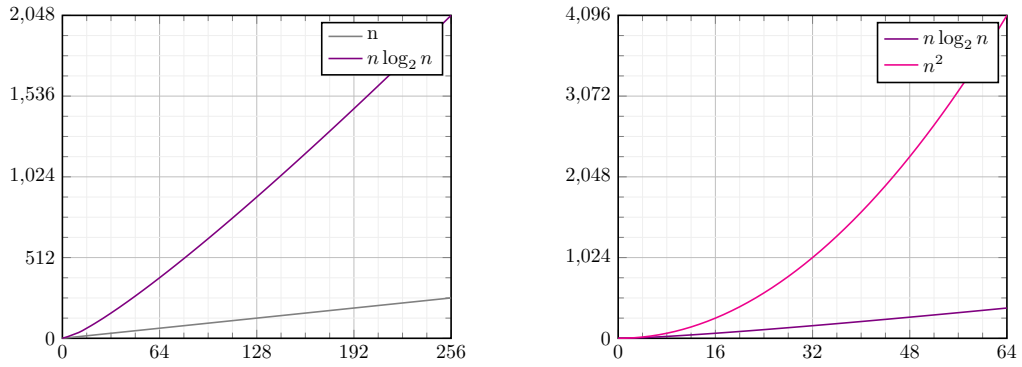
Fractional power complexity is a less common case of algorithms which reduce their input by a fractional exponent, typically 2 — i.e. $f(n) = \sqrt(n)$. A linear primality test that tests the divisibility by every number up to the square root of its input is an example of this complexity. Although both $n \log_2 n$ and $\sqrt{n}$ appear lower and very similar in the original graph, the latter grows more quickly, as can be seen in the logarithmic scale graph, but both stay several orders of magnitude below n.

Quasilinear complexity, $f(n) = n \log_2 n$ is also a common case of algorithms that reduce their domain by half as in the case of logarithmic complexity but in relation to the operations performed for each of its input elements. Sorting algorithms are an example of this complexity. Although the original graph shows this function quickly growing past the range displayed, this class of algorithms is still considered tractable as, just like for logarithmic complexity, the logarithm resembles a constant at larger values of n.

Figure 2.4a shows the relationship between the functions $n \log_2 n$ and n. While steeper, its growth is contained due to the fact that the $\log_2 n$ term has a relatively moderate growth, as we saw when we analyzed logarithmic complexity.

Polynomial complexity describes algorithms whose complexity is defined by a polynomial. Quadratic complexity, $f(n) = n^2$, is its most common type, where an algorithm performs an operation for each element of the Cartesian product of its input. Less optimized sorting algorithms, such as selection and bubble sort, are examples of this complexity.

Figure 2.4b shows the relationship between $n \log_2 n$ and n^2. While both are shown to rise quickly above n, the growth of n^2 is much more rapid (in a non-linear fashion) compared to $n \log_2 n$.



(a) n vs. $n \log_2 n$ (b) $n \log_2 n$ vs. n^2

Figure 2.4

2.3 Sorting

2.3.1 Ordering

Partial order is an ordering relation which cannot be used to order all elements of a set.

Weak partial order is a partial ordering relation which is reflexive, i.e. for relation R, $\forall x \in X, xRx$. The *less-than-or-equal* relation ($\leq$) is weak since $x \leq x$.

Strong partial order is a partial ordering relation which is irreflexive, i.e. for relation R, $\forall x \in X, \neg(xRx)$. The *less-than* relation ($<$) is strong since $\neg(x < x)$.

Total order is an ordering relation where every element can be classified as preceding or succeeding every other element.

Examples of weak partial orders that occur commonly in programming are:

- The edges of a directed acyclic graph. Ancestors in unrelated hierarchies have no relation to each other.
- The *happens-before* relation in memory ordering operations (v. 7.4, *Atomic operations*). Two operations that happen before a third are unsequenced with respect to each other.

Note that (several) total orders can always be established for finite partial orders:

- A particular topological order of a directed acyclic graph establishes a total order from the partial order described by its edges.
- A particular sequence of operations under sequential consistency establishes a total order from the partial order of memory ordering operations.

2.3.2 Equivalence

In mathematics and computer science, two concepts are involved in determining the fundamental relation between values of a given type: *equivalence* and *equality*. Equivalence (often represented as $\sim$ or $\equiv$) is described mathematically as a binary relation that is:

Reflexive $x \sim x$

Symmetric $x \sim y \implies y \sim x$

Transitive $(x \sim y) \wedge (y \sim z) \implies x \sim z$

Any relation which satisfies these properties is an *equivalence relation*. Equality is the canonical equivalence relation: x and y are *equal* iff they have the same value. Equivalence relations are interesting in both mathematics and computer science because they still apply to types which may not have a strict equality relation: it is a more general relation.

In practical terms, specifically in the context of sorting and searching algorithms, a *strict partial order* (i.e. irreflexive) is sufficient for their implementation. This allows these algorithms to be applied to types which do not have a concept of equality and simplifies their interface, since a single relation can be used for all operations instead of two. For example, the `std::sort` and `std::binary_search` algorithms (to list a few) in the C++ standard library define their semantics in terms of an equivalence relation, for which `std::less` — a generic version of the `<` operator — is the canonical implementation, establishing the following definitions:

Equality `x == y`, as dictated by `operator==`

Equivalence `!(x < y) && !(y < x)`, as dictated by `operator<`

TODO

- insertion sort
- bubble sort
- merge sort
- heapsort
- quicksort
- bogosort
- bucket sort
- radix sort

2.3.3 Selection sort

One of the simplest sorting algorithms is *selection sort*, which works by progressively sorting the left side of the range one element at a time (figure 2.5). In each iteration, the minimum value in the unsorted portion is found (*selected*) and placed at the beginning, increasing the size of the sorted range by one element. When all positions of the range have gone through this procedure, the range is sorted.

The implementation of the algorithm follows naturally from its description.

In each iteration, we guarantee the beginning of the range is already sorted, then find the minimum element, swap it with the element at the beginning of the unsorted range, and guarantee that the range is now partitioned with respect to the selected element. The definition of `min_element` is equally simple.

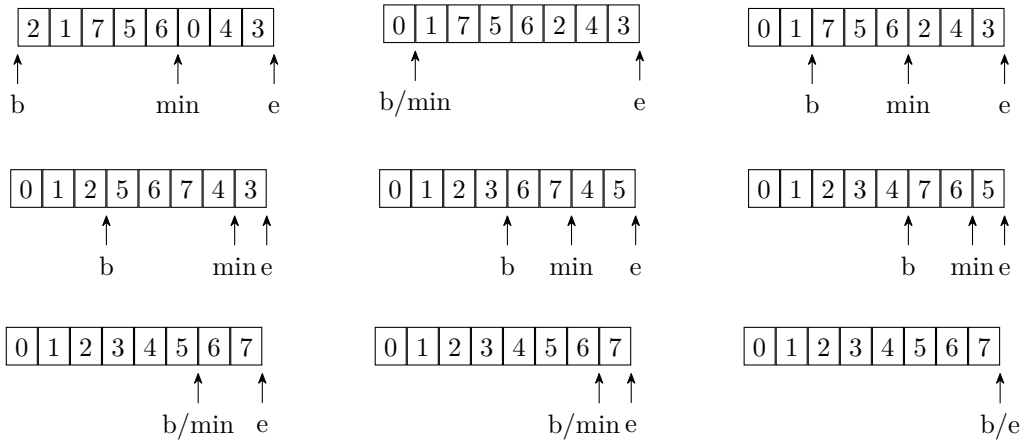


Figure 2.5: Selection sort

2.4 Binary search

Even though it may seem that examining every item in an array is required to determine whether an element is present in it, the process can be significantly improved, provided that the sequence is sorted in advance. In that case, vast portions of it can be eliminated with each comparison, a technique generally known as *divide-and-conquer*. This provides a great advantage in cases where searches are much more common than modifications to the sequence of items — i.e. where the cost of sorting the sequence can be paid once and the result used by a number of searches.

The critical aspect of the implementation is to examine the *middle* element of the sequence. If it is *less* than the desired value, the entire lower half of the array can be ignored. Conversely, if it is *greater* than the desired value, the entire upper half can be ignored. The process is then repeated until the result is found or the range becomes empty.

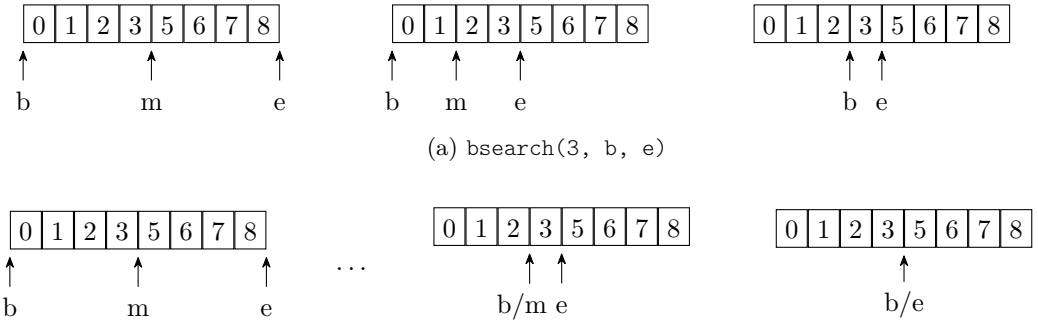
Due to this process of halving in each iteration, this algorithm is called a *binary search*, and is a dramatic improvement over a linear search. For a sequence of 2^n elements, a linear search has to examine at worst all 2^n elements. A binary search will reduce the search space to 2^{n-1} after the first iteration, 2^{n-2} after the second, and so on, until it reaches $2^0 = 1$, resulting in only $n + 1$ iterations[4].

Figure 2.6a shows one execution of a binary search on the array `[0, 1, 2, 3, 5, 6, 7, 8]` for the value `3`. The first middle value is `5`, which is greater than the desired value, so the upper portion of the array is eliminated. The next middle value is `2`, which is less than the desired value, so the lower portion of the remaining range is eliminated. The last middle value is `3`, which is the desired value, so the search ends. Figure 2.6b shows the search on the same array for the value `4`, which is similar up to the last step, where `x` is greater than `*m`, so the range becomes empty.

We start the implementation by establishing the function preconditions and the loop termination condition. Our inputs will be the value `x` to search for and the half-open interval `[b, e)` delimiting the sorted range. The terminating condition as described previously is that the range being examined becomes empty.

```
bool bsearch(int x, const int *b, const int *e) {
```

[4]Section 2.2 (*Complexity*) explores the implications of this difference in much more detail.



(a) `bsearch(3, b, e)`

(b) `bsearch(4, b, e)`

Figure 2.6: Binary search

```
    assert(is_sorted(b, e));
    const int *const ib = b, *const ie = e;
    while(b != e)
        // ...
    return false;
}
```

The loop body starts with the invariants:

```
assert(ib <= b && b <= e && e <= ie);
```

Any reduced range should be within the bounds of the original and in the correct sequence.

```
assert(!lsearch(x, ib, b) && !lsearch(x, e, ie));
```

As the input range is repeatedly reduced, we guarantee `x` is not in any of the eliminated regions — those now outside `[b, e)` but inside the original range.

```
assert(b == ib || b[-1] < x);
assert(e == ie || x < *e));
```

As `b` is moved right, it should always delimit elements that are less than `x`. Similarly, as `e` is moved left, it should always delimit elements that are greater than `x`. From these and the loop termination condition that `b == e`, we can make the following inferences about the value of `b` at the bottom of the function:

- $ib = ie \implies x \notin [ib, ie)$
  - `x` cannot be in the range since it is empty
- $b = ib \implies x \notin [ib, ie)$
  - $x < *e \implies x < *ib$
  - `x` cannot be in a sorted range if it is less than the first element
- $e = ie \implies x \notin [ib, ie)$
  - $b[-1] < x \implies e[-1] < x$

  - x cannot be in a sorted range if the last element is less than it

- otherwise $\implies$ $x \notin [ib, ie)$
  - $x < *e \implies x < *b$
  - $b[-1] < x$
  - x cannot be in a sorted range if $b[-1] < x$ and $x < b[0]$

That is, as long as the invariants are maintained, exiting the loop means x is not contained in the range. The implementation of the loop begins by determining the middle element of the range. This is done by computing its length and using integer division to halve it while rounding down.[5].

```
const int *const m = b + (size_t)(e - b) / 2;
assert(b <= m && m < e);
```

Next, the middle value is compared to x: if it is equal, the value has been found. Otherwise, the search range is modified according to the rules described previously.

```
if(*m == x)
    return true;
else if(*m < x)
    b = m + 1;
else // x < *m
    e = m;
```

We can observe that the invariants are preserved in both branches that advance to the next iteration:

- *m < x
  - $ib <= b'$, since $b' = m$, $ib <= b$, and $b <= m$.
  - $b' <= ie$, since $b' = m + 1$, $m < e$, and $e <= ie$.
  - $b'[-1] < x$, since $b' = m + 1$ and $*m < x$.

- *m > x
  - $ib <= e'$, since $e' = m$, $ib <= b$, and $b <= m$.
  - $e' <= ie$, since $e' = m$ and $m < e$.
  - $x < *e'$, since $e' = m$, $*m \neq x$, and $x < *m$.

Listing 2.1 shows the complete implementation.

```
bool bsearch(int x, const int *b, const int *e) {
    assert(is_sorted(b, e));
    const int *const ib = b, *const ie = e;
    while(b != e) {
        assert(ib <= b && b <= e && e <= ie);
        assert(!lsearch(x, ib, b) && !lsearch(x, e, ie));
```

[5]The difference is first converted to an unsigned value, since we know it is positive, so that the division by a power of two can be translated to a few simple bitwise operations. Integer division rounds values towards zero, while division using shifts rounds towards negative infinity. Converting a value to unsigned eliminates special cases in the generated machine code since those two cases have the same outcome for positive values.

```
        assert(b == ib || b[-1] < x);
        assert(e == ie || x < *e);
        const int *const m = b + (size_t)(e - b) / 2;
        if(*m == x)
            return true;
        else if(*m < x)
            b = m + 1;
        else
            e = m;
    }
    assert(ib <= b && b == e && b <= ie);
    assert(!lsearch(x, ib, ie));
    assert(b == ib || b[-1] < x);
    assert(b == ie || x < *b);
    return false;
}
```

Listing 2.1: Binary search

2.4.1 Exercises

1. Prove that the expression `m = b + (e - b) / 2` correctly calculates the middle element of a range.

2. A more common mathematical definition of the middle element would be `m = (b + e) / 2`. Could that be used in our implementation of binary search? Prove the answer and give examples.

3. Our version of binary search works and is significantly faster than linear search, but it performs two comparisons per iteration. A closely-related fact is that it will return on the first element that is found to be equal to the input.

   Often, the range can have duplicate values and it is desired to find either the first position where a value occurs or the position after its last occurrence, e.g. to find the place where a new element should be inserted to keep the range sorted. These two algorithms are called lower and upper bound. More specifically, lower bound returns the position that partitions the range into elements that are `< x`, while upper bound returns the position that partitions the range into elements that are `> v`. For the array `[0, 1, 3, 3, 3, 5, 6]`, the lower bound for value `3` is position `2`, while the upper bound is position `5`. These are the same results for the lower bound for value `2` and the upper bound for value `4`.

   Implement both versions and then rewrite the `binary_search` function in terms of `lower_bound`, all with suitable invariants.

   ```
   int *lower_bound(int x, int *b, int *e);
   int *upper_bound(int x, int *b, int *e);
   ```

Chapter 3

Data structures

3.1 Allocation

Two main dynamic allocation patterns are commonly found in software. Objects which are independent entities are given a dedicated region of memory specially allocated to store them. This is the pattern often found in dynamic, high-level languages, where even primitive objects are *boxed*, i.e. have their own distinct allocation. In compound structures composed of multiple fields, each value is simply a *reference* to its sub-objects, which also have their own dedicated allocation. Object construction in these languages is often indivisible from memory allocation, as shown in listing 3.1.

Lower-level languages such as C and C++ offer more control over the memory layout of objects. Pointers and dynamic memory allocation primitives/functions can still be used to implement the previous pattern, but often objects are *embedded* in others. An object is the concatenation of all of its constituent fields, and memory allocation and construction are usually two distinct operations, as shown in listing 3.2.

This complete control over memory allocation allows other patterns to be used. A common one is to associate extra information with an allocation: a memory allocator may store tracking information, a linked list may store previous/next pointers, an array of items may store its length, etc. A single memory block can be used both for the contained object and its associated information. Listing 3.3 shows an implementation of the latter example: an array which stores information about the items as a header prepended to the memory block.

The client interface deals solely with a `char*` which points directly to the data block. The implementation, however, actually allocates an object of type `struct array` which stores additional information about the object — the size and some undefined set of flags in this case[1]. `array_alloc` allocates a single block of memory containing both the header and the client data. This header is easily retrievable from the pointer returned to the client via the use of simple pointer arithmetic, as demonstrated in `array_size`, which uses the header information. `array_destroy` uses the same calculation to pass the correct pointer to `free` — which likely uses a similar technique for its allocation tracking data.

This structure uses the C99 *flexible array member* syntax. `data` does not occupy any actual space in the object (as demonstrated by the `static_assert`):

```
struct dedicated0 { int *i; };
struct dedicated1 { float *f; };

struct dedicated {
    struct dedicated0 *s0;
    struct dedicated1 *s1;
    double *d;
    char *name;
};

struct dedicated *dedicated_create(void) {
    const char name[] = "dedicated";
    struct dedicated *const ret = malloc(sizeof(struct dedicated));
    *ret = (struct dedicated){
        .s0 = malloc(sizeof(struct dedicated0)),
        .s1 = malloc(sizeof(struct dedicated1)),
        .d = malloc(sizeof(double)),
        .name = malloc(sizeof(name)),
    };
    *(ret->s0->i = malloc(sizeof(ret->s0->i))) = 42;
    *(ret->s1->f = malloc(sizeof(ret->s1->f))) = 43.0f;
    *ret->d = 44.0;
    strcpy(ret->name, name);
    return ret;
}
```

Listing 3.1: Structure allocation (dedicated)

```
enum { NAME_SIZE = 32 };
struct embedded0 { int i; };
struct embedded1 { float f; };

struct embedded {
    struct embedded0 s0;
    struct embedded1 s1;
    double d;
    char name[NAME_SIZE];
};

void embedded_init(struct embedded *p) {
    const char name[] = "embedded";
    *p = (struct embedded){.s0.i = 42, .s1.f = 43.0f, .d = 44.0};
    static_assert(sizeof(name) <= sizeof(p->name));
    strcpy(p->name, name);
}

struct embedded *embedded_alloc(void) {
    struct embedded *const ret = malloc(sizeof(struct embedded));
    embedded_init(ret);
    return ret;
}
```

Listing 3.2: Structure allocation (embedded)

```
// public interface
char *array_alloc(size_t n);
size_t array_size(char *p);
void array_destroy(char *p);

// private implementation
struct array {
    size_t n;
    u32 flags;
    char data[];
};
static_assert(sizeof(struct array) == 2 * sizeof(size_t));

char *array_alloc(size_t n) {
    struct array *const ret = malloc(sizeof(struct array) + n);
    *ret = (struct array){.n = n, /*.flags = ...*/};
    return ret->data;
}

size_t array_size(char *p) {
    return container_of(p, struct array, data)->n;
}

void array_destroy(char *p) {
    free(container_of(p, struct array, data));
}
```

Listing 3.3: Memory block header

it is simply convenient syntax which gives direct access to a data block of undetermined size appended to the object. Using this syntax has several advantages over other methods (such as declaring it as an array of one element[2]):

- `sizeof` accurately reports the size of the `struct` — it does not include the trailing array.
- It is not possible to use `sizeof` with the array member, it is considered as having an incomplete type. This prevents the accidental use in an attempt to incorrectly calculate the size of the array.
- Structures with trailing flexible array members cannot be placed in the middle of other `structs` or in arrays.

Prior to C99, some compilers (such as GCC) supported declaring an array of zero elements as an extension, with similar (but inferior) semantics.

[2]Section 4.4 (*Undefined behavior*) shows an example of how mistakes in this case can be subtle and catastrophic.

3.2 Linked lists

Linked lists are a simple data structure which consist of a sequence of elements and the connections between them (*links*). This contrasts with arrays, which contain just the elements themselves, where the order is implicit in the fact that elements are laid out sequentially in memory[3]. Even so, the underlying storage for the elements is immaterial: the definition of the data structure is only concerned with how elements are connected to each other.

3.2.1 Singly linked list

The simplest possible implementation of a linked list is shown in figure 3.1. A fixed-size array is used to store the elements (separated by solid lines), where each is some data and the index of the next element (denoted with arrows). The iteration process starts at the list *head*, denoted by the arrow starting on the left. After an element is processed, the next element is obtained by adding the element's "next" index to the list head. A special index value denotes the last element, the list *tail*. Iteration stops after this element is processed.

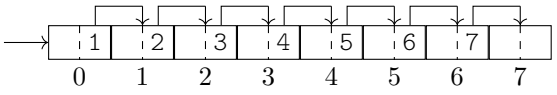


Figure 3.1: Array as linked list

Because there is a single link in each element that points forward to the next element, this type of structure is called a *singly linked list* or *forward list*. One advantage they have over arrays becomes apparent when elements are added and/or removed. When these operations are performed in an array in any position that is not the last, elements must be shift either up or down to make or fill the space for the target position. In a linked list, these operations simply manipulate the links between elements.

Figure 3.2 shows the same list after a few of these operations. The sequence of elements is now `0, 2, 4, 1, 5, 6` even though they all retain their previous location in memory.

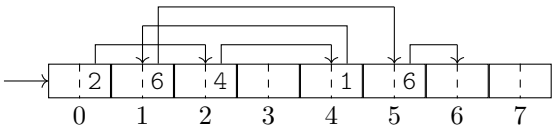


Figure 3.2: Array as linked list

Using an index as the link from one element to the next has some advantages. The array where elements are stored can be freely moved from one memory location to another, or even serialized/stored/de-serialized and the list will still retain its proper order. Indices can also be made as small as desired/possible. However, a much more common design is to use a *pointer* to the next element, with a null pointer representing the end of the list. Figure 3.3 shows the classical singly list implementation, where element indices are for clarity and do not represent relative or absolute location in memory.

[3]Subject to alignment and padding requirements, but still extrinsic and independent of the contents of each element.

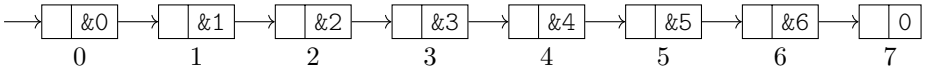


Figure 3.3: Linked list with pointers

Using pointers further dissociates the implementation of the list from the underlying storage, as elements still can but are no longer required to be contiguous and can instead be placed anywhere in memory. In this type of implementation, list elements are often referred to as *nodes*. Several allocation strategies can be used depending on the context, ranging from dynamically allocating every element individually, to using blocks or pools, to using a single array as above. These all have different advantages and disadvantages in terms of number of allocations, total memory usage, asymptotic complexity of operations, iterator invalidation, memory coherence, etc.

Since links are just pointers and can be dereferenced directly, it is no longer required to keep the list head during iteration. In fact, iteration can start at any element of the list given just its memory address. This makes linked lists a good candidate for storage in a library that exposes pointers to (sub-)objects, as the list element can remain in place when others are added/removed and can be reached easily from the pointer given to the client. Furthermore, the list head and the next pointer of each element are now the same type of value, which simplifies the implementation of many list operations, as we will see.

The simplest way to guarantee a list node can be reached from a pointer to the value it is contained in is to make the value "inherit from" it[4], either directly or via an intermediary structure. This works, since a pointer to an object of a given type can always be converted to a pointer to an object of a type it is a strict subset of, but has several disavantages. It forces the node pointer to be the first element of the value structure, which imposes a restriction on how fields can be ordered to improve structure layout and cache utilization. It also prevents a value from being placed in more than one list at a time.

The alternative, used heavily in the Linux kernel (Brown 2009), is to place the list node structure anywhere in the contained value and use the `container_of` macro function to offset the node pointer and reach the value. Since both the list head and the nodes are of type `struct node*`, they can be treated homogeneously to implement several list operations (listing 3.4).

```
struct node {
    struct node *next;
};

#define container_of(p, t, m) ((t*)((char*)(p) - offsetof(t, m)))
#define list_entry(p, t, f) ((void*)container_of(p, t, f))
#define list_for_each(h, v) for(struct node *v = (h); v; v = v->next)

void list_push_front(struct node **l, struct node *n) {
    n->next = *l;
    *l = n;
}

void list_pop_front(struct node **l) {
    *l = (*l)->next;
}
```

[4]V. section 5.3 (*Object-oriented programming*) for a detailed analysis of the patterns described in the next paragraphs.

```
struct container {
    int i;
    float f;
    struct node *head;
};

struct value {
    char *s;
    struct node *next;
};

void f(struct container *c) {
    struct node n = {0};
    list_push_front(&c->head, &n);
    list_for_each(c->head, n) {
        struct value *v = list_entry(n, struct value, next);
        use(v);
    }
    list_pop_front(&c->head);
}
```

Listing 3.4: Linked list operations

Performance

The flexibility to easily store and manipulate lists comes at a cost. As stated in section 1.4.2 (*Cache*), modern processors are optimized for sequential, predictable memory loads. Contiguous (or mostly contiguous) data structures such as arrays can be processed very quickly, for two reasons. First, more than one element can be loaded from main memory or data caches at once if they are in the same cache line. Second, since the address of the next element is always known in advance: each element is at a fixed offset from the previous: an optimizing compiler or superscalar processor can issue the memory loads for an element before the current one is processed, reducing the latency when processing each element and improving overall throughput.

In the case of a list, the address of the next element is not known until the current one is examined. This is a direct data dependency, which means fetching and processing elements has to be a serial operation. Since reordering elements merely manipulates the links and does not reorder them in memory, iterating over a list can quickly turn into random memory accesses. Certain types of list storage also have the potential to spread elements throughout memory with little coherency. For all these reasons, even though theoretically the iteration over an array and a list have the same time complexity ($O(n)$), the former is likely to be significantly faster in a modern processor.

One mitigation in certain cases is to place an explicit software prefetch instruction in the iteration code. This can in some cases mitigate the latency of the memory load and result in better interleaving of iteration and processing code[5].

```
for(struct node *n = l; prefetch(n->next), n; n = n->next)
    // ...
```

[5]V. commit `75d65a425c0163d3ec476ddc12b51087217a070c` in the Linux kernel, which removed this type of software prefetching in the common list functions, for subtle cases where this can result in *slower* code.

3.2.2 Doubly linked list

3.2.3 Exercises

1. Given the definition of a list below, implement the function `list_reverse`, which is given a list and reverses the order of the nodes in place, returning the head of the new list.

```
struct node {
    struct node *next;
    int value;
};

struct node *list_reverse(struct node *n);
```

2. Given the definition of a list in exercise 1, implement the function `list_remove`, which is given a list and a value and removes from the list the first node that contains that value, returning a pointer to the removed node or `NULL` if the value is not found.

```
struct node *list_remove(struct node **l, int value);
```

3.3 Reference counting

Brown 2009

3.4 Hash tables

- hash function
- table (addressing)
- collision resolution

```
struct entry { u32 /*h,*/ k, v; };

struct hash_table {
    struct entry v[N];
};

u32 *find(struct hash_table *t, u32 k) {
    struct entry *p = t->v;
    const struct entry *const e = p + N;
    for(; p != e; ++p)
        if(p->k == k)
            return &p->v;
    return NULL;
}
```

```
u32 hash(u32);

u32 *find(struct hash_table *t, u32 k) {
    const struct entry *const b = t->v;
    const struct entry *const e = b + N;
    struct entry *b = t->v + hash(k);
    for(; p != e; ++p)
        if(p->k == k)
            return &p->v;
    for(p = t->v; p != b; ++p)
        if(p->k == k)
            return &p->v;
    return NULL;
}
```

3.4.1 Hash functions

`https://preshing.com/20110504/hash-collision-probabilities`

- `x % N` / `x & N`
  - integers
  - random input
- `for(h = i; *p; *p++) h = h * m + *p;`
  - byte strings
    - K&R 1st ed.: `i = 0`, `m = 1`
    - K&R 2ed ed.: `i = 0`, `m = 31`
    - djb2: `i = 5381`, `m = 33` (`(h << 5) + h`)

3.4.2 Addressing

Open/closed hashing describes whether traversal escapes the table to a secondary type of storage (open) or is constrained to the table (closed).

Open/closed hashing describes whether the address in the table where an element is stored is completely determined by the hash function (closed) or varies depending on the current contents of the table (open).

3.4.3 Collisions

- External bucket for each hash value: linked list, vector, red-black tree.
- Probing
  - linear
  - quadratic
  - `https://preshing.com/20160314/leapfrog-probing`
  - ...
  - good cache locality
  - efficient for low and medium load factor

`https://preshing.com/20110603/hash-table-performance-tests`

Chapter 4

C

4.1 Declarations

Declarations in C are infamously considered complex and arcane, but follow a very simple rule: the syntax and precedence rules for the various elements that constitute it are the same as those used in expressions involving the declared variable (Ritchie 1996). Thus the following:

```
int i, *p, **p;
```

declare variables of type integer, pointer-to-integer, and pointer-to-pointer-to-integer, which are used in expressions in a similar way:

```
int x = i, y = *p, z = **p;
```

Similarly, the following:

```
int fi(), *fp(), (*pfi)(), *(*pfp)();
```

declare a function returning an integer, a function returning a pointer to integer, a pointer to function returning an integer, and a pointer to function returning a pointer to integer, which are used as:

```
int j = fi(), k = *fp(), l = (*pfi)(), m = *(*pfp)();
```

Similarly for arrays, the following:

```
int ai[10], *ap[10], (*pai)[10], *(*pap)[10];
```

declare variables of type array of 10 integers, array of ten integer pointers, pointer to array of ten integers, and pointer to array of ten integer pointers, which are used as:

```
int x = ai[0], y = *ap[0], z = (*pai)[0], w = *(*pap)[0];
```

4.1.1 Storage

Every declaration has a *storage class* associated with it, either explicitly (using a *storage class specifier*) or implicitly (context dependent). A class further subdivides in two aspects: *storage duration* and *linkage*. Storage durations, which describe the lifetime of variables, are:

Automatic duration occurs in scopes (i.e. inside functions and blocks). Variables of this type exist only inside the scope in which it they are declared and their storage is handled by the compiler on the call stack.

Static duration spans the entire execution of the program. Variables of this type have pre-allocated space in the program and are guaranteed to be initialized before the beginning of the `main` function.

Dynamic duration has its scope defined by the program. Variables of this type are allocated and deallocated at runtime using the `malloc/calloc/realloc/free` functions from the standard library (found in `<stdlib.h>`).

Thread duration was introduced in C11. Variables of this type behave similarly to *static* variables, but their duration is tied to the execution of a single thread. Each thread has its own copy of the variable, which is initialized before the thread is started.

Linkage types, which describe the parts of the program that have access to them (in the linking phase, specifically, hence their name), are:

No linkage variables are not visible to the linker at all and can only be referred to in the scope where they are declared.

Internal linkage variable are visible only in (i.e. internal to) their translation unit.

External linkage variable are visible to the entire program.

Every declaration has implicit storage duration and linkage depending on where it is placed, which can be changed with storage specifiers:

`auto` declares a variable with automatic duration and no linkage. This is the default for function parameters and local variables. It is obsolete and was only useful in ancient versions of C where untyped identifiers were assumed to be of type `int` (and even earlier in BCPL and B, where the only type was word/`int`[1]). In that context, `auto i` could be used to declare an `int` with automatic duration.

`register` behaves as `auto`, but requests that the variable be stored in a register. It was used both as a counterpart to `auto` in untyped declarations and as an optimization hint in architectures with limited registers, and is similarly obsolete. The compiler is not mandated to fulfill the request, but even if it does not this type of variable cannot have its address taken (as it may not reside in memory). This is also the only specifier which can be applied to function parameters.

`static` declares a variable with static duration. If applied to a declaration in file scope, that variable has internal linkage.

`extern` declares a variable with static duration and external linkage. This is the default for declarations in file scope (external to functions), hence its name.

[1] In the beginning was the word.

thread_local declares a variable with thread duration[2]. It only applies to objects, either in file or block scope. For the latter, it must be combined with `static` or `extern` to determine the linkage (this type of declaration always has thread duration).

inline

One specifier related to linkage is `inline`. Its primary purpose is as a hint to the compiler that it may be advantageous to inline a function at its call site. With the advances in the optimizer in compilers, this usage is virtually obsolete. It has, however, maintained its other property: an `inline` function may be defined in more than one translation unit, making the definition effectively `static`. This is necessary since this type of function is usually placed in a header file included in many translation units.

Note that the behavior of `inline` in C is subtly different from C++, from where it was adopted: there may be one definition of an `inline` function with external linkage. This results in convoluted rules to determine the address of the function and what types of `static` variables a function can declare or has access to.

4.2 Preprocessor

A preprocessing directive is a line starting with the **#** character followed by one of the predefined directive names. The directive spans the entire line up to the new-line character. Any amount of white space can precede or succeed the **#** character, but any other type of preceding character makes that line not a preprocessing directive. This applies even if macro expansion (described later) removes the preceding characters or transforms them into white space, so this is one way to write an `escape` macro, useful if a file must be processed more than once:

```
#define ESCAPE
ESCAPE#define X
```

4.2.1 Macro replacement

Macro replacement is a text substitution facility of the C preprocessor. As other preprocessor directives, it takes place before the text is processed by the compiler as C code. Macro definitions are created via the **#define** directive and have one of the following forms:

```
#define O object-like macro
#define F() function-like macro
```

In both cases, `#define` is followed by an identifier: the *name* of the macro (`O` and `F` in this case). Any subsequent mention of the macro's name in the source code is replaced with the *replacement text*, the portion of the macro definition which follows its name (`object-like macro` and `function-like macro` in this case).

Object-like macros (such as `O`) are defined when no parentheses immediately follow the name. Mentions of the name in the source code are replaced with the replacement text, which may be empty.

[2]The actual keyword is `_Thread_local`, to comply with the standard's rules for backwards compatibility of identifiers in the global scope. The `<threads.h>` header defines `thread_local` as an object macro which is replaced with `_Thread_local`.

```
0 // expands to ‘object-like macro‘
```

Function-like macros (such as `F`) are defined when the identifier is immediately followed by parentheses containing zero or more comma-separated identifiers. Mentions of the name in the source code which are also followed by parentheses (just like in a regular function call) are replaced with the replacement text. Before the text is replaced, each mention of each parameter in the replacement text is replaced by the argument supplied at the point where the macro was invoked.

```
F() // expands to ‘function-like macro‘
```

`#undef`

There is a single namespace for all types of macros. A name cannot be redefined if a definition already exists[3], but a definition can be removed using the **`#undef`** directive. Following it, the name can again be defined as any type of macro. A macro only affects token processing between its corresponding `#define` and `#undef` directives (or until the end of the translation unit, in case no `#undef` directive is present).

The `#` and `##` operators

In the replacement text of function-like macros, any parameter name preceded by the **`#`** character is replaced by a textual version of itself, instead of literally. That is, it is effectively placed inside double quotes and characters are escaped as necessary to produce a valid string literal. Additionally, any sequence of white space characters (including new-line characters) not inside string literals is collapsed to a single space character and preceding and succeeding white space characters are removed.

Any parameter name preceded or succeeded by **`##`** is replaced by the tokens that form the argument. Then, any **`##`** in the replacement text (not originating from the arguments) is removed and its preceding and succeeding tokens are concatenated. Any two tokens can be concatenated as long as the result is also a valid token.

Variable arguments

The ellipsis (`...`) can appear as the last parameter of a function-like macro. This type of macro can be invoked with any number of arguments, as long as enough values are provided for the other parameters, if they exist. Trailing arguments that do not correspond to the named parameters form the *variable arguments*, which include the separating comma characters. Inside the replacement text, these arguments can be expanded using the **`__VA_ARGS__`** identifier.

Recursive expansion

A powerful aspect of macro expansion is that whenever a replacement occurs, macros are recursively expanded according to the following rules:

[3] Multiple definitions of a macro are allowed provided they are all equivalent, which is defined as having the same number, order, spelling, and white-space separation (all white-space separations are considered identical) in their replacement list. In addition, function-like macros are identical only if they have the same number and spelling of arguments.

- When a parameter (including `__VA_ARGS__`) is replaced in the replacement text of a function-like macro, it is examined for macro names, which are fully expanded before the replacement is performed.
- For both object- and function-like macros, the replacement text, along with the subsequent tokens in the source file, is reexamined after expansion for more macro names.
- The concatenation resulting from the application of the ## operator happens before the replacement is reexamined, and the result is available for further expansion.

```
#define str(x) #x
#define f(x) x

// ‘str‘ is expanded before ‘f‘
str(f(0))
// "f(0)"

#define hash_hash # ## #
#define str(x) #x
#define in_between(x) str(x)
#define join(x, y) in_between(x hash_hash y)

// the ‘##‘ produced by expanding ‘hash_hash‘ is not the ‘##‘ operator
join(x, y)
// in_between(x hash_hash y)
// in_between(x ## y)
// str(x ## y)
// "x ## y"
```

4.2.2 Macro metaprogramming

```
#define PROMOTE(x) (+(x))


#define PROMOTE_TO(x, y) (1 ? (x) : (y))


#define IS_SIGNED(x) (PROMOTE(-1, x) < PROMOTE(1, x))
```

4.3 Expressions

4.3.1 Pointers

`NULL` pointer not necessarily zero value (e.g. C++ member pointers)

> [T]he name of the null pointer is “0”, but the name of the null pointer is called “NULL” (and we’re not sure what the null pointer is).

Summit 1995

4.3.2 Compound literals

C99 introduced a type of expression which declares a temporary value of aggregate types (i.e. array, `struct`, or `union`), called a *compound literal*. The syntax is visually similar to a cast operation:

```
// Array literal.
// Note: could also be declared directly as an array in this case.
// int a[] = {0, 1, 2, 3};
int *a = (int[]){0, 1, 2, 3};
// `struct` literal.
struct S { int i; float f; };
struct S s = (struct S){.i = 42, .f = 3.1415f};
// `union` literal.
union U { int i; float f; };
union U u = (union U){.f = 3.1415f};
```

This type of literal can be used whenever a value would be defined for a single use. It has the advantage that it does not create an identifier for the value which leaks into the surrounding scope. In fact, there is no way to refer to the value unless it is given a name explicitly, either via an assignment or function parameter.

```
struct S {};
void f(struct S);
// Only `f` has access to the temporary value, via its parameter.
f((struct S){});
```

The syntax for the declaration of compound literals is similar to that of aggregate initialization of a temporary value in C++:

```
void f(int[]);
f(std::array{0, 1, 2, 3}.data());

struct S { int i; float f; };
void f(S);
f(S{.i = 42, .f = 3.1415f});

union U { int i; float f; };
void f(U);
f(U{.f = 3.1415f});
```

There is a very important semantic difference between the two languages, however: the scope/lifetime of the temporary object when this type of declaration occurs at block scope. Because of the strict rules in C++ regarding construction and destruction of objects (which can have arbitrary side effects), the lifetime follows the same rules as any other type of temporary value, i.e. it extends only to the end of the full expression where the object is used. In the previous examples, the objects are destroyed and inaccessible after the semicolons following the expressions that declare them.

In C, the temporary object is assigned a duration according to its enclosing scope. A declaration at file scope assigns it static storage duration. This is virtually the same as when aggregate initialization is used in the definition of an object with static storage duration in C++. However, for declarations at block scope, C assigns automatic storage duration just as if the object had been defined as a local variable prior to the expression where it is used: its scope spans the entire block where it is declared. This makes code like the following possible, where a temporary array is created as part of an expression and used throughout the surrounding block:

```c
#define FMT(buffer, fmt, ...) fmt_str((buffer), (fmt), __VA_ARGS__)

inline static char *fmt_str(
    char *restrict buffer, const char *restrict fmt, ...
) {
    va_list args;
    va_start(args, fmt);
    vsprintf(buffer, fmt, args);
    va_end(args);
    return buffer;
}

int main(void) {
    char *const s = FMT((char[128]){0}, "%s %d %f", "str", 42, 3.1415);
    puts(s);
}
```

An attempt to use the similar C++ syntax would not work as expected, as the lifetime of the object ends as soon as the assignment is concluded:

```c
int main(void) {
    char *s = FMT(std::array<char, 128>{}, "%s %d %f", "str", 42, 3.1415);
    // temporary array (and ‘s‘, by extension) no longer valid here
    std::puts(s);
}
```

4.3.3 Anachronisms

The legacy of C's predecessors — BCPL and B — is apparent in many aspects of the language, leading to surprises and "infelicities" (Ritchie 1996). Section 4.1.1 (*Storage*) already mentioned the `auto` and `register` keywords, a relic of BCPL's untyped variables (the former later resurrected in C++ for automatic type inference).

`&&` and `||` precedence

Both of those languages predate the `&&` and `||` operators. They instead relied on context-dependent interpretation of the single-character `&` and `|` operators, which were the familiar bitwise operations in regular expressions but had the effect of the logical operators inside an `if` statement. This resulted in expressions such as:

```c
if(x == y & z)
    // ...
```

which tests whether `x` is equal to `y` and `z` is not zero. This explains the surprising precedence rules for the `||`, `&&`, `&`, `|`, and `==` operators (listed here in increasing order of precedence), and why parentheses are often required in expressions such as the following:

```c
if((x & y) == z)
    // ...
```

where `&` would have lower precedence than `==` otherwise, usually not what is intended.

`struct` members

The V6 Unix source code contains several occurrences of a curious pattern:

```
#define SW 0177570

struct { int integ; };

if(SW->integ == 0)
    // ...
```

This syntax is a consequence of several aspects of early compilers (Lions 1977). Casts did not exist at the time, there was a single namespace for structure members, and there were no checks for the type of structure used on the left-hand side of the `->` operator. This meant that the expression above was equivalent to the following in modern C:

```
if(*(int*)((char*)SW + 0) == 0)
    // ...
```

i.e. the value of `SW` was treated as the memory address of a(n anonymous) structure containing an integer member at offset `0`.

4.4 Undefined behavior

The standard has precise definitions of the observable behavior of most operations which constitute a C program, but explicitly excludes some of them, categorized as:

Undefined behavior completely invalidates a program. The language assigns no meaning to the operation and any program which includes it is not required to have any meaningful behavior at all.

Unspecified behavior can result in different outcomes in different platforms or even in different usages within the same program.

Implementation-defined behavior has the same semantics as unspecified behavior, but the implementation must document what the outcomes are.

Locale-specific behavior varies according to calls to the standard library function `setlocale`.

The primary reason for the first three categories is to guarantee the efficiency and portability of C code across platforms which have different implementations of those operations, as discussed in detail in the following sections. Defining their behavior would penalize those whose instructions do not natively support the chosen behavior and force compilers to emit extra code to compensate.

4.4.1 Integer arithmetic

As discussed in section 1.1 (*Integers*), arithmetic involving unsigned numbers is simple, while different representations, with different advantages and disadvantages, can be used for signed numbers. Two's complement with modular arithmetic is the de facto standard, used in most architectures and implementations[4], but other representations are still in use, and for a long period of time there was not a clear dominance.

[4]In fact, it is the only representation allowed by the C++20 standard, planned to be adopted by the C23 standard as well.

Even among platforms with the same underlying integer representation, the behavior of operations can differ. As an example, the semantics of integer *overflow* and *underflow* — when the result of an operation cannot be represented by its corresponding type — are among the most significant in the context of undefined operations. The behavior of specific platforms varies independently of their integer representation, and include:

- The operation is computed with *saturating arithmetic*, i.e. results are bound within a predefined range. Values which exceed the range are adjusted to the minimum/maximum of the range.
- The operation is computed with *modular arithmetic*, i.e. results outside the range are replaced by their equivalent according to the modulus of the type. Condition code bits in the CPU's status registers are usually set when this happens.
- The operation generates an *interrupt*, either initiated by the hardware or by software through the use of special compiler arguments (e.g. `-ftrapv` in GCC/Clang).

For these reasons, low-level programming languages such as C, which operate close to the hardware platform, have strict rules for unsigned arithmetic, but leave room for implementations to define many aspects of signed arithmetic semantics. These operations are classified as having either implementation-defined or undefined behavior. Doing otherwise would require extra code in implementations whose native operations did not match those dictated by the standard[5]. Continuing with the overflow/underflow examples, the standard states that:

- `UINT_MAX + 1` must be `0u`
- `0u - 1` must be `UINT_MAX`
- `INT_MAX + 1` results in undefined behavior (not `INT_MIN`)
- `INT_MIN - 1` results in undefined behavior (not `INT_MAX`)
- `CHAR_MAX + 1` and `SHRT_MAX + 1`, however, are interesting cases which demonstrate the surprising complexity of the C type system. Because of integer promotion rules, the result is either:
  - the expected value, but with type `int`, or
  - undefined due to overflow

  depending on whether `INT_MAX` is larger than `CHAR_MAX` and `SHORT_MAX` (respectively) on a given platform — i.e. whether the `char` or `short` constant is promoted to `int` prior to the addition.
- The unary minus (`-`) operator can cause overflow in two's complement implementations (where the value zero takes one value out of the positive range) if applied to the minimum value of a signed type. The result is one greater than the maximum value, which is outside the range of the type and equal to the minimum value in modular arithmetic. This results in surprising equalities such as `-INT_MIN == INT_MIN`.

[5]Note that by imbuing unsigned types with both "positive" and "modular" semantics, the standard constrains somewhat the optimization advantages in platforms whose native instructions do not perform modular arithmetic. A type system in which these concepts were orthogonal could allow the generation of code that is closer to the machine capabilities in those cases.

Bitwise shift operations are another example where not all expressions have defined behavior, again due to divergent hardware behavior. Shifting signed values is restricted by the integer representation just as in the case of overflows. In the case of left shifts, the resulting value has to be valid for its type, otherwise the behavior is undefined. The behavior of right shifts, in contrast, is implementation-defined: it may perform regular logical shift or arithmetic shift (or something else entirely, possibly). The specific case of shifting values — even unsigned — by an amount that is equal to or larger than the type's width in bits is another example of undefined behavior to accommodate different implementations:

- 32-bit shifts on x86 are truncated to 5 bits, so
  - `x << 32` = `x << 64` = `x << 0` = x
- 32-bit shifts on PowerPC are truncated to 6 bits, so:
  - `x << 32` = `x << 0` = x, but
  - `x << 64` = 0
- 32-bit shifts on ARM are truncated to 8 bits, so
  - `x << 32` = `x << 64` = 0

In all the cases presented in this section, "well defined" should not be confused with "correct" behavior. It is entirely possible for code to use exclusively well-defined operations and still be incorrect. One famous example (Dietz et al. 2012) from GCC is shown in listing 4.1. As indicated by the comment, the function allocates a vector of `N` elements. `struct rtvec_def`, however, uses the common over-allocation technique of declaring a trailing array of size `1` of (effectively) `rtunions`, so that `sizeof(struct rtvec_def)` already includes one element. The `n - 1` expression then calculates the correct number of extra elements which need to be allocated. This works as expected except when `n == 0`, in which case the ultimate behavior is very surprising, if well-defined:

```
/* Allocate a zeroed rtx vector of N elements */
rtvec rtvec_alloc(int n) {
    rtvec rt;
    int i;
    rt = (rtvec)obstack_alloc(
        rtl_obstack,
        sizeof(struct rtvec_def) + ((n - 1) * sizeof(rtunion)));
    // ...
    return rt ;
}
```

Listing 4.1: Wraparound in an allocation function in GCC

1. `n - 1` results in `-1`, a signed integer value.
2. `sizeof` expressions have the type `size_t`, however, so the result is converted prior to the multiplication to `(size_t)-1` (i.e. `SIZE_MAX`, a very large number) due to the rules of integer promotion in arithmetic expressions.

3. The result of the multiplication exceeds the range of the type, but that is valid since it is `size_t`. The result under modular arithmetic is `SIZE_MAX - sizeof(rtunion) - 1`.

4. The result of the addition again exceeds the range of the type. The final value of the expression is `sizeof(struct rtvec_def) - sizeof(rtunion)`.

Even though there is no undefined or even implementation-defined behavior involved in any of the sub-expressions, the ultimate result is in no way what was expected by the programmer: an allocation of insufficient size for a `struct rtvec_def`[6].

4.4.2 Pointer arithmetic

Arithmetic involving at least one pointer value is only defined if the pointers involved are themselves valid. A pointer can only be formed to existing objects and arrays, or to one element past the end of an array. Operations are similarly only defined if pointer operands and results point to valid sub-elements of a common structural type (arrays, `structs`, `unions`). This rule implicitly — in most cases — invalidates overflow/underflow when integer operands are involved and pointers are implemented as memory indices, since it implies the resulting pointer is not part of the original object. Subtraction with two pointer operands not only requires valid values under the same rules, but is also only defined if the resulting offset fits its type: `ptrdiff_t`, a signed integer type with range [`PTRDIFF_MIN`,`PTRDIFF_MAX`].

4.4.3 Optimizations

While some of the portability concerns described so far in this section have diminished due to the increasing adoption of common representations, undefined behavior is still very relevant in a different but related area: compiler optimization. In this context, an equally valid conception of undefined behavior is as a *restricted* or *narrow contract* (Carruth 2016) offered by the programming language. The reduced set of guarantees for signed integers, for example, compared to their unsigned counterparts gives the compiler more freedom to transform the operations involving them in a way that facilitates other optimizations. Examples of the types of optimizations allowed by undefined behavior in C are [7]:

- Expression simplification
  - `x * 2 / 2` can be replaced with `x` if the multiplication is assumed to not overflow.
- Loop simplification
  - `for(int i = 0; i <= N; ++i)` can be assumed to always execute exactly `N + 1` times (`N == INT_MAX` would result in an infinite loop).
  - `for(int i = b; i <= e; i += n)` can be assumed to always terminate (`INT_MAX - n < e` would result in an infinite loop for some values of `n` and `INT_MAX`).
- 32- to 64-bit integer promotion

[6] Although do note that modular arithmetic guaranteed that the semantic intent of the complete expression (allocate a `struct rtvec_def` with `n` — i.e. zero — elements) was maintained even in the presence of multiple out-of-range values. The outcome of an invalid C object is unfortunate but independent of the mathematical interpretation of the expression.

[7] While some may seem nonsensical, they may be the result of the expansion of inline functions and macros and never appear directly in source code. The simplified expressions can also in turn enable further optimization.

  - In 64-bit code which mixes 32- and 64-bit values, 32-bit signed integers can be assumed to not overflow (and require modular arithmetic), "promoted" to 64-bit, and used in expressions with other 64-bit values[8].

- Strict aliasing

  - Pointers of different types are treated as if they were `restrict` with respect to each other (except for `char` pointers, which can alias any other pointer).

Another example from Carruth 2016 which demonstrates one of these optimizations is code adapted from the `bzip` compression program (listing 4.2). The purpose of the code (comparing blocks of bytes when sorting) is not as relevant as how it operates, the types it uses, and the machine code that is generated[9]. A block of bytes is supplied to the function along with a pair of indices. The first part of the function, shown in the list, compares subsequent elements pointed to by the given indices, incrementing each by one position after each comparison.

```
bool mainGtU(char *p, unsigned i0, unsigned i1) {
    char c0, c1;
    if((c0 = p[i0++]) != (c1 = p[i1++])) return c0 > c1;
    if((c0 = p[i0++]) != (c1 = p[i1++])) return c0 > c1;
    // ...
}
```

Listing 4.2: `bzip`'s `mainGtU`

On the x86-64 architecture, the block pointer and the indices are, respectively, 64- and 32-bit values. Indexing the block involves offsetting the pointer by the value of the indices, a simple addition operation (a scaling operation would also be involved if `sizeof(*p)` was not `1`). There exists an addressing mode dedicated to such situations, but examining the (abbreviated) generated machine code reveals it is not used in this case:

```
# ...
mov cl, byte ptr [rdi + rcx]
cmp byte ptr [rdi + rax], cl
jne .LBB0_4
lea eax, [rdx + 1]
lea ecx, [rsi + 1]
mov al, byte ptr [rdi + rax]
cmp byte ptr [rdi + rcx], al
jne .LBB0_4
```

[8] A tangential topic is the curious decision of 64-bit platforms to maintain `int` as a 32-bit type, instead of continuing the progression and making its width equal the register size of the platform. It is often assumed (and, also often, in a depreciative manner) that this happened simply because of backward compatibility, but that is just one of the factors that influenced that decision (The Open Group 1997). In summary:

  - The 2^{32} range of 32-bit integers was already enough for most applications. Expanding it was much less advantageous compared to, for example, the previous expansion from 16 bits ($2^{16} = 65536$).
  - The integer categories in C would not be enough to express all desirable widths in common platforms (this predated the C99 fixed width types). With an 8-bit `char` and a 64-bit `int`, the only intermediate type, `short`, would have to be either 16- or 32-bit, and there would be no type left for the other size.
  - Integer promotion rules would convert values with a rank less than `int` to 64-bit in all arithmetic expressions.

[9] Generated with Clang 13.0.1 and `-O2`. Interestingly, GCC does not perform this optimization.

```
lea eax, [rdx + 2]
lea ecx, [rsi + 2]
mov al, byte ptr [rdi + rax]
cmp byte ptr [rdi + rcx], al
jne .LBB0_4
# ...
```

The indices, initially in the `rdx` and `rsi` registers, are instead incremented using the `lea` instruction and then stored in the (32-bit portions of the) `eax` and `ecx` registers. The expected machine code in this case would be the following, which can be generated by simply changing the indices from unsigned to signed variables of the same size (i.e. `int`):

```
# ...
mov dl, byte ptr [rdi + rcx]
cmp byte ptr [rdi + rax], dl
jne .LBB0_4
mov dl, byte ptr [rcx + rdi + 1]
cmp byte ptr [rax + rdi + 1], dl
jne .LBB0_4
mov dl, byte ptr [rcx + rdi + 2]
cmp byte ptr [rax + rdi + 2], dl
jne .LBB0_4
# ...
```

Here, the values remain in their original registers and used in the pointer/offset/immediate-value x86 addressing mode directly. The reason for this difference is the restricted contract of signed integers: because the compiler has the guarantee that the signed indices will not overflow, it can treat them effectively as 64-bit values. The same is not possible with unsigned indices since they must obey the 2^{32} modulo arithmetic in case of overflow. Using the (64-bit) addressing mode in that case would change the semantics of the operations.

4.4.4 C++

It is worth at this point to compare certain C++ concepts to their C equivalents. Even though they can be implemented in C, the stricter requirements, especially regarding undefined behavior, in C++ often result in better code generation.

Converting an object to one of its bases may involve a pointer adjustment if the base is not the single root of the hierarchy (in which case it is pointer-interconvertible). This conversion usually requires an extra check for the `nullptr`, since it must yield another `nullptr`. In architectures where pointers are memory offsets and `nullptr` is the literal value `zero`, unconditionally adding the base offset would not give the correct result. Listing 4.3 shows the generated code for this case.

```
struct S { int s; };
struct T { int t; };
struct U : S, T { int u; };

void f(S*), g(T*);
void h(U *p) { f(p); g(p); }
```

Listing 4.3: Base pointer conversion

```
_Z1hP1U:
    push   rbp
    mov    rbp, rdi
    call   _Z1fP1S@PLT
    lea    rax, 4[rbp]
    test   rbp, rbp
    cmovne rbp, rax
    mov    rdi, rbp
    pop    rbp
    jmp    _Z1gP1T@PLT
```

An implementation in C would need similar checks to replicate this functionality. However, C++ also states that a member function call through a `nullptr` results in undefined behavior. This allows the compiler to eliminate the check for member function calls (listing 4.4).

```
struct S { int s; void g(void); };
struct T { int t; void g(void); };
struct U : S, T { int u; };

void h(U *p) { p->f(); p->g(); }
```

Listing 4.4: Member function optimization

```
_Z1hP1U:
    push rbx
    mov  rbx, rdi
    call _ZN1S2fsEv@PLT
    lea  rdi, 4[rbx]
    pop  rbx
    jmp _ZN1T2ftEv@PLT
```

Chapter 5

C++

5.1 Calling conventions

The standard for function parameters that have no special qualification is *pass-by-value*: a function that declares a parameter of type `T` receives a *copy* of the argument. An alternative is *pass-by-reference*, where either a pointer or a reference to the object is passed.

Reference arguments avoid copying objects and are potentially a faster way to pass objects that are large and/or expensive to copy. Value arguments, however, have several advantages and are usually preferred for types that are relatively inexpensive to copy. They are often stored in registries (according to the ABI, see below), avoiding any sort of memory access.

References are also subject to *aliasing*, where an object is referred to by more than one name. A compiler has to be conservative when manipulating references and members of structures passed by reference and reload values from memory when there is the possibility that they are aliased. Because value parameters are not located in memory, the compiler can aggressively optimize access to them with the assumption that the parameter name is the only reference to its value.

The particular manner in which objects are passed between functions depends on the *Application Binary Interface* of a particular platform.

5.1.1 System V / Itanium ABI

These are the C and C++ (respectively) ABIs adopted by most 64-bit Unix derivatives (Linux, FreeBSD, Solaris, macOS, etc., v. Matz et al. 2012 and *Itanium C++ ABI* 2017). In these conventions, integral (including pointers) and floating-point values are passed in registers. Six registers are designated for the first six integral parameters (`rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`), eight for the first eight floating-point parameters (`xmm0` to `xmm7`), in both cases in the same order they appear in the function prototype. Any remaining parameters are passed via the call stack (listing 5.1).

For aggregate types (`structs`), the Itanium ABI discriminates based on what it defines as *non-trivial for the purpose of calls*: types whose constructors and/or destructor are non-trivial, i.e. not generated by the compiler. A type is considered *trivial*, as indicated by the type trait `std::is_trivial`, if it has only trivial constructors and destructor.

`structs` of this category are always passed by reference to a function, if the formal parameter is declared to be by-value. The compiler will insert the operations to place arguments on the stack (*stack spilling*) if necessary.

```
#include <stdint.h>

void f(
    int8_t i8,
    int16_t i16,
    int32_t i32,
    int64_t i64,
    const char *s,
    const int *p,
    float f,
    double d,
    int64_t i64_0,
    int64_t i64_1);

void g(void) {
    int p = 5;
    f(
        0, 1, 2, 3, "", &p,
        6, 7, 8, 9);
}
```

```
g:
    sub   rsp, 24
    mov   ecx, 3                      ; i64
    mov   edx, 2                      ; i32
    xor   edi, edi                    ; i8
    mov   DWORD PTR 12[rsp], 5        ; p
    movsd xmm1, QWORD PTR .LC0[rip]   ; d
    lea   r8, .LC1[rip]               ; s
    mov   esi, 1                      ; i16
    push  9                           ; i64_1
    movss xmm0, DWORD PTR .LC2[rip]   ; f
    push  8                           ; i64_0
    lea   r9, 28[rsp]                 ; &p
    call  f@PLT
    add   rsp, 40
    ret
.LC0:
        .long   0
        .long   1075576832
        .align 4
.LC1:
        .string ""
.LC2:
        .long   1086324736
```

Figure 5.1: Scalar arguments passed by value

Trivial `struct` types are treated mostly as if each member were a separate argument, following the same rules for register designation. Additionally, members may be packed if more than one fit into a single register (listing 5.2).

```
#include <stdint.h>

struct S { int32_t x, y; };

void f(struct S _);
void g() { f((struct S){0, 1}); }
```

```
g:
    movabs rdi, 4294967296
    jmp    f@PLT
```

Figure 5.2: Structural arguments passed by value

5.2 Metaprogramming

5.2.1 Template instantiation

Function, class, and variable templates by themselves are not usable directly in code: they must first be instantiated, i.e. the specific definition — of which there can be many due to overloads and specializations — has to be determined and its template parameters replaced. Instantiation can happen in two ways:

Explicit instantion is an expression whose entire purpose is to generate a particular template instantiation. It can be used to precisely control the placement of the code generated from the instantiation.

Implicit instantiation happens when the template is used in an expression that would require a definition. The template is instantiated from the context it is used in and the resulting definition is used in the expression.

Both explicit and implicit instantiation of function, class, and variable templates operate under the same general rules and consist of the following stages:

Name lookup associates the referenced name with one or more declarations. In the case of overloaded functions, this list can also include non-template functions.

Template argument deduction determines the types or values (for type and non-type parameters, respectively) of the template arguments according to the instantiating expression and the template declaration.

Overload resolution / specialization selection determines which member of the overload set or which specialization is used.

Template argument substitution replaces every occurrence of each parameter in the selected template definition with its corresponding argument.

Simple examples of implicit instantiations are:

```
template<typename T> void f(T);
f(0); // instantiates f<int>
template<typename T> struct S {};
S<int>{}; // instantiates S<int>
template<typename T> auto v = T{};
v<int>; // instantiates v<int>
```

Of the four stages mentioned above, only name lookup and substitution apply in all cases: none of the examples above have specializations and only `f` deduces its arguments. It is always possible to specify template arguments explicitly: `f<int>(0)` would be an equivalent instantiation and require no deduction. Partial argument substitution is performed on the viable declarations and deduction, when it happens, is applied to the result. For function templates, an explicit argument list, even if empty (e.g. `f<>(0)`), eliminates non-template functions from the overload set.

5.2.2 SFINAE

One fundamental aspect of how template instantiation happens, in particular the selection of an overload or specialization in particular, is that invalid expressions are temporarily allowed to be formed. The technical term used by the standard for such cases is *ill-formed*, and the result is a *substitution failure*. Instead of generating a compilation error, as an ill-formed program would in any other context, only the substitution fails and the overload or specialization is simply discarded. This follows naturally from the way overloading and specialization work, but has profound consequences and forms the basis of template metaprogramming: *substitution failure is not an error* (SFINAE).

To understand why it is necessary in those contexts, consider the following example, where a parameter of one of the functions templates in an overload set is a reference. The first call matches the first declaration, with `T` as `int`, while the second matches the second declaration, with `T` as `void`. However, both also match the other declaration in each case, and both substitutions are ill-formed: the first generates an instantiation with unresolved parameters, while the second

generates a reference to `void`. If SFINAE did not apply, both instantiations would ultimately be invalid. It does, so the program is valid and calls the first and second functions in the overload set, as expected.

```
template<typename T> void f(const T&);
template<typename T> void f(...);

f(0), f<void>();
```

Substitution failures are only ignored in what is called the *immediate context* of the instantiation, the definition of which has been expanded throughout the history of C++. It includes:

- Types used in the function prototype.
- Types used in the template parameter declarations.
- Expressions used in the function prototype (since C++11).
- Expressions used in the template parameter declarations (since C++11).
- Expressions used in the `explicit` specifier (since C++20).

Most notably missing in this list is the body of the function: substitution errors in its definition are not ignored. The same is true for any further template instantiations that occur in the immediate context, so in the example below the correct version of `f` is called while the call to `g` generates a compilation error, even though they are functionally equivalent. This is because the substitution error happens in the instantiation of `S`, which is not considered the immediate context of the substitution in `g`.

```
template<typename T, typename U = T::type> void f(void);
template<typename T> void f(...);

template<typename T> struct S { using type = T::type; };
template<typename T, typename U = S<T>::type> void g(void);
template<typename T> void g(...);

struct T {};
f<T>(), g<T>();
```

The examples so far demonstrated substitution failures involving type expressions. The prototypical application of this pattern in the standard library is the `std::enable_if` type, shown in listing 5.1. It is a type metafunction whose parameters are a boolean value and a type. The specialization for `true` as the first argument simply returns that type unchanged. The base case — whose argument can only be `false` — on the other hand declares no return type at all. The usual convenience template alias is also defined.

The utility of `enable_if` might be questionable at first, but not in the context of SFINAE. Because the `type` member only exists when the predicate is true, expressions such as `enable_if<b>::type` and `enable_if_t<b>` are ill-formed when `b` is `true`. If used during template substitution, this face can be exploited to discard an otherwise viable candidate based on the value of the boolean predicate. Consider the following code, which declares specialized versions of a function `f` for integral and floating-point types:

```
template<typename T> std::enable_if_t<std::integral<T>> f(T);
template<typename T> std::enable_if_t<std::floating_point<T>> f(T);

f(0), f(0.0);
```

```
template<bool, typename T = void>
struct enable_if {};

template<typename T>
struct enable_if<true, T> : std::type_identity<T> {};

template<bool b, typename T = void>
using enable_if_t = enable_if<b, T>::type;
```

Listing 5.1: `enable_if`

Value categories are mutually exclusive, so only one of the predicates will ever be true. In that case, the expansion of `enable_if_t` will result in `void`, and the corresponding version of `f` will have the type `void(void)`. The other version will be ill-formed, since the `enable_if` predicate will be false, and will be discarded. The end result is an unambiguous instantiation depending on the predicate, i.e. the value category of `T`.

Since C++11, another type of substitution failure is also ignored: substitution in expressions. This was due to the introduction of the `decltype` specifier, which greatly increased the contexts in which expressions can be used in declarations. Consider for example the code below, where a `decltype` specifier is used in the declaration of a function's return type. The expression, and as a consequence the entire function prototype, will only be valid if `f` is a callable object and `args` is a valid sequence of arguments for it.

```
template<typename F, typename ...Args>
auto f(F &&f, Args &&...args) -> decltype(FWD(f)(FWD(args)...));
```

The standard library provides the `std::invocable` concept and the `std::is_invocable` type trait (with a few variations), which have a similar purpose and can be implemented using this technique, as partially demonstrated in listing 5.2. The `is_invocable_impl` overload set uses two familiar devices: a base case which uses variable arguments and a `decltype` specifier which tests the validity of an expression. However, here the comma operator is used to dissociate the expression from the actual return type. The expression must still be well-formed for this declaration to be considered valid, but the resulting type will be that of the last expression — here simply `std::true_type`, following the semantic rules of the comma operator.

`decltype` is used again in the primary definition, a simple alias template, demonstrating another common pattern. The two `is_invocable_impl` prototypes are never actually defined: they are used purely as a syntactical mechanism to choose between two types — `std::false_type` and `std::true_type`. This combination of an overload set — possibly including function templates — used exclusively in a `decltype` specifier is very common in metaprogramming.

The prototypical application of this pattern in the standard library is the `declval` "function", another utility that is impressive in its conciseness and usefulness, shown in listing 5.3. Similar to `is_invocable_impl` above, this function is never actually defined: its entire purpose is to be used in *unevaluated contexts* (such as `sizeof` and `decltype` expressions). Its advantage over a simpler expression such as `T{}` is that it works even if the type is not default-constructible. This is because it is restricted to unevaluated contexts, so it does not need to be defined and avoids having to form the expression required to actually initialize an object of the given type.

Another deceptively simple standard library utility, introduced in C++17, is `void_t`. Its definition, shown in listing 5.4, barely has more characters than `declval`'s. The code below shows an application. It uses yet another SFINAE technique: forming a type or expression in

```
std::false_type is_invocable_impl(...);

template<typename F, typename ...Args>
auto is_invocable_impl(F &&f, Args &&...args)
    -> decltype(FWD(f)(FWD(args)...), std::true_type{});

template<typename F, typename ...Args>
using is_invocable =
    decltype(is_invocable_impl(std::declval<F>(), std::declval<Args>()...));

template<typename F, typename ...Args>
inline constexpr bool is_invocable_v = is_invocable<F, Args...>::value;

static_assert(is_invocable_v<int(float, void*), float, void*>);
static_assert(!is_invocable_v<int(float, void*), void*, float>);
```

Listing 5.2: `is_invocable`

```
template<typename T> T &&declval(void);
```

Listing 5.3: `declval`

```
template<typename...> using void_t = void;
```

Listing 5.4: `void_t`

the partial specialization of a class template. If the substitution is ill-formed, the specialization is discarded just as if it were an invalid member of an overload set[1].

```
template<typename T, typename = void>
struct referenceable : std::false_type {};

template<typename T>
struct referenceable<T, std::void_t<T&>> : std::true_type {};

template<typename T>
inline constexpr bool referenceable_v = referenceable<T>::value;

static_assert(!referenceable_v<void>);
static_assert(referenceable_v<int>);
static_assert(referenceable_v<int&>);
```

`void_t` in this context serves as a very convenient and concise place for the predicate type expression. If `T&` is a valid expression (i.e. if `T` can have a reference specifier attached to it, essentially equivalent to `!std::is_void<T>`), the substitution succeeds and the `std::void_t` instantiation becomes simply a complicated type alias for `void`, which matches the primary template definition (the default template argument is fixed before partial specializations are analyzed, so this `void` has to match it). If the substitution generates an invalid type (i.e. `void&`), it fails and the specialization is discarded.

[1] In reality, this application of SFINAE is not mentioned in the standard (v. CWG issue 2054). Practically, however, it is implemented by every C++ compiler and widely used, being even a part of the future *Library Fundamentals v2* Technical Specification.

5.2.3 Fold expressions

```
template<typename T>
struct last<types<T>> {
    using type = T;
};

template<typename T, typename ...Ts>
struct types_last<types<T, Ts...>> {
    using type = last_t<types<Ts...>>
};



template<typename ...Ts>
struct last<types<Ts...>> {
    using type = first_t<decltype((..., types<Ts>{}))>;
};



template<typename T, typename R>
using test = std::is_same_v<types_last<T>, R>;

static_assert(test<types<int>, int>);
static_assert(test<types<void, int>, int);
static_assert(test<types<void, int&>, int&);
static_assert(test<types<void, int&&>, int&&);
static_assert(test<types<void, const volatile int&>, const volatile int&);



// generated with:
// $ seq -f 'S<\%.0f>,' "$(((1 << 13) - 1))" -1 0
template<int> struct S {};
static_assert(
    test<
        types<
            S<8191>,
            S<8190>,
            // ...
            S<0>
        >,
        S<0>>);
```

GCC and Clang reject this version of the code with `S<1>` as the second argument in ∼0.25s and ∼0.3s, respectively[2], while using the recursive implementation takes ∼31s and ∼36s.

5.3 Object-oriented programming

The term *object* generally means a (possibly empty) data type and associated functions — called *methods* — that operate on those data: state and behavior[3]. The former can be as simple as the C `struct`. Methods are often (but may not be) available using the same member access syntax.

[2]Interestingly, GCC takes ∼0.7s to accept the original assertion. Tested with GCC 11.1.0 and Clang 13.0.0.

[3]When coding, some programmers mix functional and imperative. Others prefer not to: they believe in the separation of Church and state. -- Robert Sewell

5.3.1 Method dispatch

The first distinctive characteristic of object-oriented languages and systems is that the behavior exhibited by an object is based on its "type", which may not be known until the point where a method is called at runtime.

A common pattern in C is to augment `structs` with members which are function pointers taking as an argument the structure itself, by value or pointer (listing 5.5, figure 5.3a). This pattern is very much similar to dynamic languages such as Lua, where each object can have a distinct combination of method implementations.

While useful in some cases, this pattern is also needlessly wasteful when this flexibility is not necessary and many instances of the `struct` (i.e. objects) have the same values for their function pointer members. The method pointers are usually moved to a separate structure so that multiple objects can refer to the unique combinations of method implementations (listing 5.6, figure 5.3b). This method-only structure is usually called a *virtual function table* (abbreviated as *vtable*).

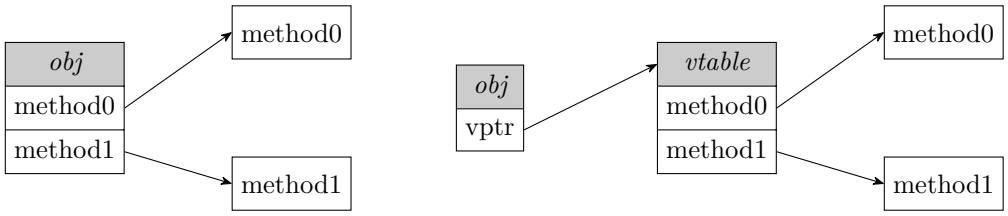
The benefit of this organization is the reduced storage requirement: each unique list of function pointers is now stored once instead of being repeated in every object. This comes at the cost of an additional pointer dereference, which can be alleviated by CPU caches for frequently-used tables.

```
struct object {
    void (*method0)(struct object*);
    void (*method1)(struct object*);
    int data;
};

void f(struct object *o) {
    o->method0(o);
}
```

Listing 5.5: Object with method pointers

```
struct object {
    struct vtable *vptr;
    int data;
};

struct vtable {
    void (*method0)(struct object*);
    void (*method1)(struct object*);
};

void f(struct object *o) {
    o->vptr->method0(o);
}
```

Listing 5.6: Object with virtual table pointer



(a) Object with method pointers

(b) Object with virtual table pointer

Figure 5.3: Storage options for methods

Throughout this introductory section, excerpts from the Linux kernel will be used as examples of a large C code base with several object-oriented concepts[4]. `struct file` and `struct file_operations` (listing 5.7) are an example of the latter pattern: the former contains it is

responsible for implementing the various types of files that a Unix/Linux system might have, while providing the generic interface available through all file descriptors.

```
// include/linux/fs.h (with minor edits)
struct file {
    struct path f_path;
    struct inode *f_inode;
    const struct file_operations *f_op;
    // ...
};

// ...

struct file_operations {
    loff_t (*llseek)(struct file*, loff_t, int);
    ssize_t (*read)(struct file*, char*, size_t, loff_t*);
    ssize_t (*write)(struct file*, const char*, size_t, loff_t*);
    int (*mmap)(struct file*, struct vm_area_struct*);
    int (*open)(struct inode*, struct file*);
    int (*flush)(struct file*, fl_owner_t id);
    // ...
};
```

Listing 5.7: `struct file_operations`

The structure should look familiar from the previous example. `operations` is the usual suffix given to virtual table structures in Linux. One advantage of the manual implementation of object-oriented concepts is the unlimited power to adapt the design, unimpeded by any assumptions or constraints a more rigid language implementation might have.

`file_operations` is a small example with respect to its members: contrary to a virtual table in traditional object-oriented languages, many of its method pointers can be `NULL`. An unimplemented method in one of those languages would (through different mechanisms depending on the particular language) default to the implementation of its parent classes. A low level implementation is free to interpret a `NULL` value however suits it, and this interpretation can vary per call site. In the case of `file_operations`, the meaning of a `NULL` value varies for each operation:

- `llseek`: A default operation is performed (modifying the position counter in `struct file`, in this case). This is analogous to a default implementation in object-oriented languages, but note that the code is inlined in the caller and not executed by an indirect call.
- `read`: Signals that this object does not support the operation. `read(2)` calls will return `EINVAL`.
- `ioctl`/`unlocked_ioctl`: An interface is being transitioned from one method to the other. Both can coexist, although only one is ever set for any given object. When all instantiations have been changed, the old method is removed.
- `aio_read`/`aio_write`: A similar case in that only one of the `read`/`write` methods or these are set: the asynchronous I/O API is such that one method can be implemented in terms of the other. In this case, however, there is no transition and the two types of methods are expected to exist indefinitely.

[4] V. Brown 2011 for a thorough discussion.

- Finally, new operations added will default to `NULL` (v. section 4.3.2, *Compound literals*), so testing for that value can allow for incremental development. A caller can immediately start using the new method when available while it can be progressively add to objects.

The defaulting of a method's implementation can also be effected in the initialization of the virtual table object, instead of by treating the `NULL` value specially. A useful pattern in C is to provide a macro that is placed at the beginning of a compound literal (listing 5.8).

```
#define DEFAULTS \
    .method0 = method0_default, \
    .method1 = method1_default

const struct vtable vtable = {
    DEFAULTS,
    .method0 = method0,
};
```

Listing 5.8: Compound literal default macro

This has the advantage that the calling code can avoid a branch when calling the method, which will make the code shorter and faster when skipping the indirect function call is not a common occurrence.

Another possibility when implementing virtual tables is to store more than method pointers in them. Some object-oriented languages do this internally to support some level of reflection (e.g. *run time type information*, RTTI, in C++). The type of information stored in this way is usually related to the class of objects that share the virtual table. `md_personality` is one such example in Linux (listing 5.9).

```
// drivers/md/md.h (with minor edits)
struct md_personality {
    char *name;
    int level;
    struct list_head list;
    struct module *owner;
    bool (*make_request)(struct mddev *mddev, struct bio *bio);
    int (*run)(struct mddev *mddev);
    int (*start)(struct mddev *mddev);
    void (*free)(struct mddev *mddev, void *priv);
    void (*status)(struct seq_file *seq, struct mddev *mddev);
    // ...
};
```

Listing 5.9: `struct md_personality`

Here we see additional members that provide information such as a descriptive name and owning module, and intrusive linked list pointers related to the registration of these virtual tables.

5.3.2 Inheritance

Another aspect present in many object-oriented languages is the ability for objects to inherit data and behavior from parent "types". The relationship between each level in the hierarchy with regards to how it operates on the complete set of data and behavior can vary, from complete

separation where every level deals with only its own members, to more relaxed scenarios where different levels cooperate to implement the full object functionality.

The simplest implementation of data inheritance that is useful is the C `union`: each possible combination of extra members is given its own field in the union and users of the object decide somehow (via a discriminant tag, based on context, etc.) which field to operate on (listing 5.10).

```
struct object {
    int common;
    union {
        struct object_impl0 impl0;
        struct object_impl1 impl1;
        struct object_impl2 impl2;
    } u;
};

struct object_impl0 { int data; };
struct object_impl1 { float data; };
struct object_impl2 { char *data; };
```

obj		
common		
impl0	*impl1*	*impl2*
data	data	data

Listing 5.10: Data inheritance with `union`

This implementation is simple and may be preferred when a heterogeneous contiguous list of objects is required, but has significant problems:

- The structure has to be changed anytime a new implementation is added.
- Unions always occupy the size of its largest member, so every object will be as large as the largest "child" object.

One alternative solution fixes both problems, but introduces some of its own. A generic `void` pointer can be added to the structure, allowing client code to set it to whatever value it wants. This is seen in `struct seq_file` in the kernel, which is a utility for pseudo text files exposed in the file system by the kernel, such as some files in `/proc`. `seq_open` (listing 5.11) is used to initialize a `seq_file` based on the `struct file` we have seen before. `file` has such a `void*` member called `private`, which is used to store an object of the "derived" `struct seq_file`. It also makes use of a virtual table called `struct seq_operations`.

`seq_file` in turn has its own `private_data` member, which is used by whatever code implements a particular path. The utility function `seq_open_private` further simplifies this process by `kzalloc`ing a block of memory of the desired, calling `seq_open` setting its private pointer, and returning that pointer to the caller. `kallsyms_open`, for example, which implements `/proc/kallsyms`, attaches a `struct kallsyms_iter` to the `seq_file` it creates (figure 5.4).

While it solves the specific problem of the worst-case size for the structure, there is now an extra pointer member, and the data attached to that pointer potentially needs one to point back to the common structure as well. It also introduces the problem of where the extra data are stored, creating an additional allocation in the worst case, and an additional memory dereference when it has to be loaded.

Another solution to those problems is to invert the relationship between the two structures: each "derived" object includes the base object somewhere in its definition (listing 5.12). Since the offset of the member is a fixed-size constant known at compile time, the outer object can be found from a pointer to the inner with simple pointer arithmetic (listing 5.13).

```c
// fs/seq_file.c (with minor edits)
int seq_open(struct file *file, const struct seq_operations *op) {
    struct seq_file *p = file->private_data;
    if(!p) {
        p = kmalloc(sizeof(*p), GFP_KERNEL);
        if(!p)
            return -ENOMEM;
        file->private_data = p;
    }
    memset(p, 0, sizeof(*p));
    mutex_init(&p->lock);
    p->op = op;
    // ...
}
```

Listing 5.11: seq_open

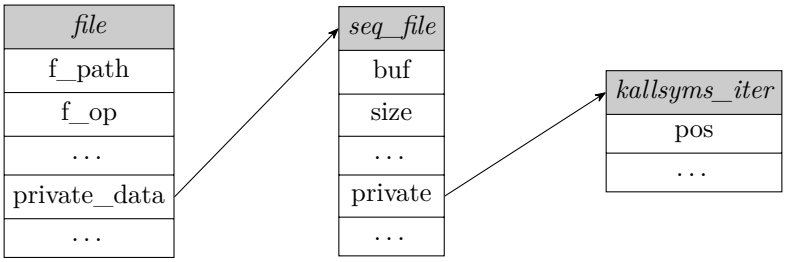


Figure 5.4: kallsyms

In particular, placing the base object at the beginning of the derived object has several beneficial properties. Both C and C++ guarantee that members appearing in the same order at the beginning of two unrelated structures will have the same offset and that a pointer to an object can be safely converted to a pointer to the type of its first member[5]. This makes it ideal to implement single inheritance as in C++ in this manner (listing 5.14). Since all base objects are prefixes of their children, the base address of the derived is also a valid pointer to all of its bases, and are guaranteed to be inter-convertible.

5.3.3 Multiple inheritance

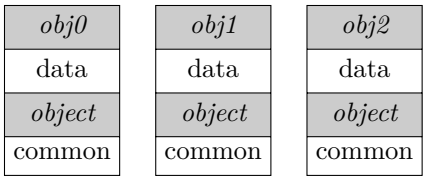
All types of inheritance shown so far involved a one-to-one relation between base and derived objects. Yet, sometimes it is desirable for an object to have multiple bases, to combine methods and/or data from multiple other objects in one, and it is not desirable to make either of those a base of the other(s). With a large number of combinations, it may also be prohibitive to instantiate all of the possible combinations a priori.

[5]C++ 20 introduced type traits specifically this verification: std::is_pointer_interconvertible_base_of, std::is_pointer_interconvertible_with_class, and std::is_corresponding_member.

```c
struct a { int data; };
struct b { int data; };
struct c { int data; };

struct d /* : a, b, c */ {
    struct a a;
    struct b b;
    struct c c;
    int data;
};

void f(d*);
struct a *pa = &obj.a;
// or pa = (struct *a)&obj
struct b *pb = &obj.b;
struct c *pc = &obj.c;
struct d *pd = &obj;
f((struct d*)pa);
f(container_of(struct d, b, pb));
f(container_of(struct d, c, pc));
f(pd);
```

Listing 5.15: Adjustment in pointer conversion

```
struct object { int common; };

struct object_impl0 {
    int data;
    struct object obj;
};

struct object_impl1 {
    float data;
    struct object obj;
};

struct object_impl2 {
    char *data;
    struct object obj;
};
```



Listing 5.12: Data inheritance with embedded objects

```
#define container_of(p, type, member) \
    ((type*)((char*)(p) - offsetof(type, member)))

void f(struct object *o0) {
    struct object_impl0 *o = container_of(o0, struct object_impl0, obj);
    // ...
}
```

Listing 5.13: `container_of`

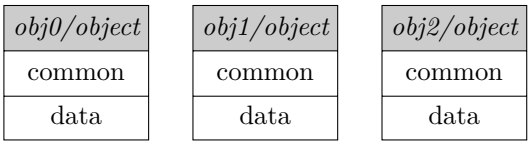
While inheritance of data and behavior was kept largely separate in previous sections, the implementation of multiple inheritance is more complex and often combines both[6]. Consider the hierarchy in listing 5.15.

A pointer to `d` and a pointer its `a` portion are inter-convertible, since `a` is a common prefix of both. A pointer to `d` can obviously be used unchanged. Obtaining a pointer to `d` from a pointer to its `b` or `c` portion is not so trivial. The `container_of` macro can be used here since the original type of the object is known, but in general it could point to any object that contains a `b` or `c` at any offset, unknown at compile time. While the compile-time knowledge can be used for optimization (a process called *devirtualization*), this must often be a runtime calculation.

The problem is fundamentally introduced by multiple inheritance: under regular single inheritance, all objects in the hierarchy are inter-convertible, so the same pointer can be passed to functions at any level. In this example, however, `d` could have a mixture of functions from `a`, `b`, and `c`, and could override any of them (or one of its subclasses could). Each method expects to receive a pointer to its corresponding type, and it is in general impossible to determine the

[6]The implementation in this section is based on Stroustrup 1989.

```
struct object { int common; };

struct object_impl0 {
    struct object obj;
    int data;
};

struct object_impl1 {
    struct object obj;
    float data;
};

struct object_impl2 {
    struct object obj;
    char *data;
};

void f(struct object*);

f(&(struct object_impl0){});
f(&(struct object_impl1){});
f(&(struct object_impl2){});
```



Listing 5.14: C++ data inheritance

required offset for derived objects at compile time.

Since these offsets are a property of the type and are used in function calls, an obvious place to store them is the table of function pointers. Each function can have a different offset, so each entry will consist of the function pointer and the object offset. Calling a function no longer involves applying a fixed offset, the on in the virtual table must be used. In addition, now the object must have more than one table (one for each base) and the table for a particular base type differs depending on the type of the derived object (as they will have different offsets). The root of the hierarchy can still share a table with the derived object, so in this example a `d` object requires three tables (one for `a`/`d`, one for `b`, and one for `c`). Listings 5.16 and 5.17 show the complete implementation.

This solution works, but comes at a price. `a`, `b`, and `c` now have larger virtual tables since they could be used in a class hierarchy, even though they don't themselves use multiple inheritance. Both the space and computation overhead is paid by all classes, regardless of their use. To arrive at a better solution, it can be observed that only functions in `d`'s table (the object which uses multiple inheritance) really need the application of the offset: they are zero in all other tables. The calculation can be moved into a function prelude, which can in turn be used as the address in the virtual table (listing 5.18). This restores the simple layout of tables for bases that do not use multiple inheritance: they can simply contain the original function pointers.

```
#define vptr_shared(b, d) \
    union { \
        struct b b; \
        const struct d ## _vtbl *vptr; \
    }

#define call(t, vf, p) call_impl((p)->t.vptr->vf, (p))
#define vcall(vf, p) call_impl((p)->vptr->vf, (p))
#define call_impl(vf, p) (((vf).f)(voff(&(vf), (p))))

struct vfn { void (*f)(); ptrdiff_t off; };

void *voff(const struct vfn *v, void *p) {
    return (char*)p + v->off;
}

struct a_vtbl { const struct vfn af0, af1; };
struct b_vtbl { const struct vfn bf0, bf1; };
struct c_vtbl { const struct vfn cf0, cf1; };
struct d_vtbl { const struct a_vtbl a; const struct vfn df; };

struct a { const struct a_vtbl *vptr; const char *data; };
struct b { const struct b_vtbl *vptr; const char *data; };
struct c { const struct c_vtbl *vptr; const char *data; };
struct d { vptr_shared(a, d); struct b b; struct c c; const char *data; };

void af0(struct a*), bf0(struct b*), cf0(struct c*);
void af1(struct a*), bf1(struct b*), cf1(struct c*);

const struct a_vtbl a_vtbl = {.af0 = {.f = af0}, .af1 = {.f = af1}};
const struct b_vtbl b_vtbl = {.bf0 = {.f = bf0}, .bf1 = {.f = bf1}};
const struct c_vtbl c_vtbl = {.cf0 = {.f = cf0}, .cf1 = {.f = cf1}};

void d_af1(struct d*), d_bf1(struct d*), d_cf1(struct d*), df(struct d*);

const struct d_vtbl d_vtbl = {
    .a = {
        .af0 = {.f = af0},
        .af1 = {.f = d_af1, .off = -(ptrdiff_t)offsetof(struct d, a)},
    },
    .df = {.f = df},
};

const struct b_vtbl db_vtbl = {
    .bf0 = {.f = bf0},
    .bf1 = {.f = d_bf1, .off = -(ptrdiff_t)offsetof(struct d, b)},
};

const struct c_vtbl dc_vtbl = {
    .cf0 = {.f = cf0},
    .cf1 = {.f = d_cf1, .off = -(ptrdiff_t)offsetof(struct d, c)},
};
```

Listing 5.16: Multiple inheritance function calls

```
int main(void) {
    struct d d = {
        .a = {.vptr = &d_vtbl.a, .data = "da"},
        .b = {.vptr = &db_vtbl, .data = "db"},
        .c = {.vptr = &dc_vtbl, .data = "dc"},
        .data = "d",
    };
    struct a a = {.vptr = &a_vtbl, .data = "a"}, *ap = &d.a;
    struct b b = {.vptr = &b_vtbl, .data = "b"}, *bp = &d.b;
    struct c c = {.vptr = &c_vtbl, .data = "c"}, *cp = &d.c;
    struct d *dp = &d;
    af0(&a), af1(&a);
    bf0(&b), bf1(&b);
    cf0(&c), cf1(&c);
    df(&d);
    vcall(af0, ap), call(a, af0, dp);
    vcall(af1, ap), call(a, af1, dp);
    vcall(bf0, bp), call(b, bf0, dp),
    vcall(bf1, bp), call(b, bf1, dp);
    vcall(cf0, cp), call(c, cf0, dp),
    vcall(cf1, cp), call(c, cf1, dp);
    vcall(df, dp);
}
```

Listing 5.17: Multiple inheritance function calls (cont.)

```
#define call(t, vf, p) (((p)->t.vptr->vf)((p)))
#define vcall(vf, p) (((p)->vptr->vf)((p)))

struct a_vtbl { void (*af0)(), (*af1)(); };
struct b_vtbl { void (*bf0)(), (*bf1)(); };
struct c_vtbl { void (*cf0)(), (*cf1)(); };
struct d_vtbl { const struct a_vtbl a; void (*df)(); };

struct a { const struct a_vtbl *vptr; const char *data; };
struct b { const struct b_vtbl *vptr; const char *data; };
struct c { const struct c_vtbl *vptr; const char *data; };
struct d { vptr_shared(a, d); struct b b; struct c c; const char *data; };

void af0(struct a*), bf0(struct b*), cf0(struct c*);
void af1(struct a*), bf1(struct b*), cf1(struct c*);

const struct a_vtbl a_vtbl = {.af0 = af0, .af1 = af1};
const struct b_vtbl b_vtbl = {.bf0 = bf0, .bf1 = bf1};
const struct c_vtbl c_vtbl = {.cf0 = cf0, .cf1 = cf1};

void d_af1(struct d*), d_bf1(struct d*), d_cf1(struct d*), df(struct d*);
void d_pre_af1(struct d *p) { d_af1(container_of(p, struct d, a)); }
void d_pre_bf1(struct d *p) { d_bf1(container_of(p, struct d, b)); }
void d_pre_cf1(struct d *p) { d_cf1(container_of(p, struct d, c)); }

const struct d_vtbl d_vtbl = {
    .a = {.af0 = af0, .af1 = d_pre_af1},
    .df = df,
};

const struct b_vtbl db_vtbl = {.bf0 = bf0, .bf1 = d_pre_bf1};
const struct c_vtbl dc_vtbl = {.cf0 = cf0, .cf1 = d_pre_cf1};
```

Listing 5.18: Multiple inheritance function calls (optimized)

Chapter 6

Unix

6.1 Linux

6.1.1 Capabilities

Traditionally, a *privileged* process is Unix systems is one with effective UID of 0. Many system administrator operations in the kernel can only be done by this type of process. While very simple to implement, this scheme has the disadvantage that the security model is binary: a process has either *no* or *all* privileges.

Capabilities are an improvement present in Linux and some other Unix derivatives. Initially part of the unsuccessful POSIX 1003.1e draft standard, capabilities break the single "privileged" status of a process[1]into many parts (v. `capabilities(7)` for the full list). The kernel then checks for a specific capability instead of an UID; a privileged process can drop all capabilities it does not need and greatly reduce the possibility that it could be used as an attack vector in a compromised system.

As a simple example, a program which sets the system time, such as `date -set ...`, needs only the `CAP_SYS_TIME` capability. On a traditional Unix system without capabilities, that program would need to have an effective UID of 0, meaning it could in principle perform *any* privileged operation. There is no reason for it to be able to change mount points, network configuration, or perform any of the many other system administration operations: capabilities allow a process to restrict its privileges to just those necessary for its function.

While capabilities have been referred to as "a set" of "a process", that is a simplification of the actual Linux implementation. They can be assigned to processes and files, and there is not one but four sets.

Sets

Ultimately, capabilities are sets of 64 bits associated with each process or file. Three sets are associated with each, while two other are specific to processes:

- The *effective* set is the one used by the kernel in its permission checks.
- The *permitted* set serves, in conjunction with the effective set, an analogous function to the real/effective UIDs: a process can temporarily reduce its effective (and inheritable) set

[1]Throughout this section, "process" is used for simplicity, but note that capabilities are a thread (i.e. task) attribute.

and later restore it to any subset of its permitted set. Removing a capability from this set is irreversible (by the process, but see capabilities attached to files below).

- The *inheritable* set determines which capabilities are preserved by a privileged process after an `exec` system call[2]. Only privileged processes (`CAP_SETPCAP`) can add capabilities to this set.

- The *ambient* set determines which capabilities are preserved by an unprivileged process after an `exec` system call. It is a subset of the intersection of the permitted and inheritable sets.

- The *bounding* set ultimately limits all capabilities which a process can acquire. This set is maintained when an `exec` system call is performed. The `init` process starts with a full set, and at any time a privileged process (`CAP_SETPCAP`) can irreversibly remove a capability from its set — and from all of its descendants as a consequence.

The sets associated with a process can be examined via the `proc` file system:

```
$ p=$(pgrep --full --oldest systemd-timesyncd)
$ grep Cap /proc/$p/status
CapInh: 0000000002000000
CapPrm: 0000000002000000
CapEff: 0000000002000000
CapBnd: 0000000002000000
CapAmb: 0000000002000000
$ capsh --decode=0000000002000000
0x0000000002000000=cap_sys_time
$ getpcaps $p
2461: cap_sys_time=eip
```

Files

Sets can be associated with files using `security` extended attributes to grant new capabilities to processes when it is used in an `exec` system call:

- The *permitted* set is used to grant new capabilities.

- The *inheritable* set further limits a process' own inheritable set when the preserved capabilities are calculated.

- The *effective* set similarly limits a process' own effective set.

In addition, the ambient set of a process is cleared if the executed file has any of: set-user-ID, set-group-ID, or file capabilities. The complete equations for the new permitted (P'_p), effective (P'_e), inheritable (P'_i), ambient (P'_a), and bounding (P'_b) sets based on the previous (P) and file (F) sets are:

[2]Capabilities are not preserved across `exec` calls because the call may fail and may require privileges that are not required by the new executable (e.g. `CAP_DAC_OVERRIDE`).

$$
\begin{aligned}
P'_p &= (P_i \cap F_i) \cup (P_b \cap F_p) \cup P_a \\
P'_a &= \texttt{!file\_privileged} \times P_a \\
P'_e &= F_e \ ?\ P'_p : P'_a \\
P'_i &= P_p \\
P'_b &= P_b
\end{aligned}
$$

UID transitions

The following changes, designed to adhere to the traditional Unix semantics, are made to the capability sets of a process when it changes UID:

- If either the real, effective, or saved set-user IDs are zero but none are after the transition, the permitted, effective, and ambient sets are cleared.
- If the effective UID transitions from zero to non-zero, the effective set is cleared. The reverse transition repopulates it using the values in the permitted set.
- Similarly, if the file system UID transitions from zero to non-zero or the reverse, the same changes are made, but restricted to a few file-system-related capabilities (v `capabilities(7)`).

The first rule can be disabled using a mechanism called `securebits` (also available via `libcap(3)`)[3]:

```
prctl(PR_SET_SECUREBITS, prctl(PR_GET_SECUREBITS) | SECBIT_KEEP_CAPS);
```

A privileged process can use this to execute a program using an unprivileged user with a few select capabilities:

```
cap_set_secbits(cap_get_secbits() | SECBIT_KEEP_CAPS);
setresgid(100, 100, 100);
setresuid(1000, 1000, 1000);
cap_t c = cap_get_proc();
cap_clear(c);
const cap_value_t v = CAP_NET_ADMIN;
cap_set_flag(c, CAP_PERMITTED, 1, &v, CAP_SET);
cap_set_flag(c, CAP_INHERITABLE, 1, &v, CAP_SET);
cap_set_proc(c);
cap_free(c);
cap_set_ambient(CAP_NET_ADMIN, CAP_SET);
execv(/* ... */);
```

[3] V. Edge 2015 for a discussion of the introduction of ambient capabilities to enable unprivileged processes to inherit capabilities.

Chapter 7

Concurrency

7.1 Amdahl's law

When analyzing the potential improvement in execution time provided by parallel threads of execution, the runtime of the problem under consideration has to be separated in two categories: one that can be executed in parallel and one that has to be serialized. Given a proportion p between the two, the total execution time T is:

$$T = (1 - p)T + pT$$

The serial portion cannot by definition be accelerated with additional threads of execution. Any speedup factor s is applied only to the p factor, such that the new execution time $T(s)$ is:

$$\begin{aligned} T(s) &= (1 - p)T + \frac{p}{s}T \\ &= (1 - p + \frac{p}{s})T \end{aligned}$$

Thus, the speedup S for any process is:

$$\begin{aligned} S &= \frac{T}{T(s)} \\ &= \frac{T}{(1 - p + \frac{p}{s})T} \\ &= \frac{1}{1 - p + \frac{p}{s}} \end{aligned}$$

This general formulation is *Amdahl's law* for the latency speedup of an improvement to a system as a function of the proportion of that system to which the improvement applies[1]. Figure 7.1 shows the plot of this relation for different values of p and s. It is evident that, at any

level of parallelism, even a small portion of serial work is a significant bottleneck. The inability to use the additional processing power is more pronounced the higher the processor count, and eventually any amount of serial work becomes a limitation, a vertical asymptote:

$$S = \frac{1}{1 - p + \frac{p}{s}}$$
$$\lim_{s \to \infty} S = \frac{1}{1 - p}$$

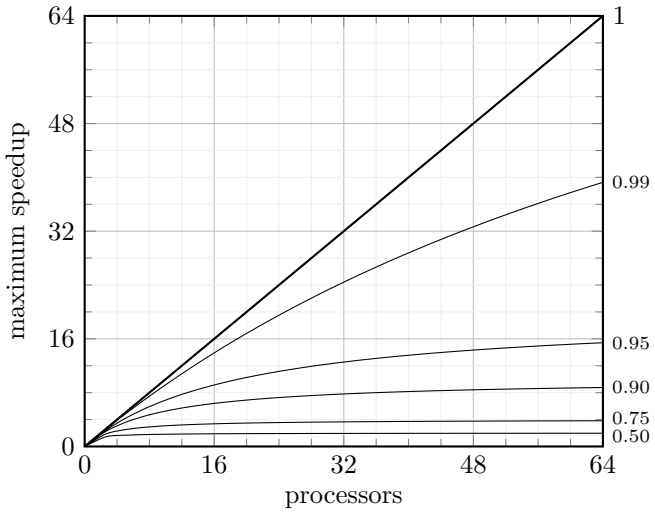


Figure 7.1: Amdahl's law applied to parallel processors

The fundamental rule on which this law is predicated is very simple: a program will never be faster than its serial portion, no matter how many resources are added. A program that is 10% serial has a maximum speedup of 10 ($\frac{1}{1-0.1} = 10$), as the 10% will never be affected by the addition of resources. I.e. the theoretical maximum speedup of a process is always limited by its serial portion.

7.2 Memory consistency

Single-threaded programs have a relatively simple sequential model: instructions are executed in the order they appear in the source code, subject to the control flow defined by the programming language. However, this simple model does not describe how the program is ultimately executed in most computer architectures.

While it is true that the *effect* of the program is *as if* execution had proceeded in that manner, compilers and processors are free to carry out instructions in any way that produces a result

[1]This general definition applies to any kind of system, including serial processes. Seen in another way, Amdahl's law states the impact of improving one of the phases of a process given the proportion of the total execution time it represents.

indistinguishable from what would be produced strictly under that model (usually called the *as-if rule*, v. *The as-if rule* n.d.). As the discrepancy between processor and memory access speeds became increasingly larger under *Moore's law*, hardware designers started incorporating more complex mechanisms to hide the long latency of memory operations, and source code became increasingly abstracted from the actual sequence of operations performed by the machine.

For single-threaded programs, this fact is barely observable in most respects. Other than secondary effects such as differences in execution time, by definition the as-if rule does not perturb their execution. However, as Moore's law began to falter, increasingly performance improvements are achieved by the use of additional processor cores, instead of the previous regular increase in transistor count. This has made understanding and managing the compiler and hardware transformations a critical factor in writing correct and efficient software.

When multiple threads of execution are involved, a *memory model* must be defined to govern the interaction of different threads observing and mutating the same region of memory. The memory system itself is effectively atomic, from the perspective of the execution threads, making the models serializable. The two aspects of a memory model are:

Memory coherence constrains the observable behavior of reads and writes to the *same* memory location. Operations can be thought of as occurring in a single timeline and all processors agree on that order. No constraint is imposed on the *order* of the operations, only on the coherence across processors.

Memory consistency constrains the observable behavior of reads and writes to *different* memory locations with respect to each other and other instructions.

The four orderings governed by a memory consistency model are:

W $\rightarrow$ **R** write must complete before subsequent read

R $\rightarrow$ **R** read must complete before subsequent read

R $\rightarrow$ **W** read must complete before subsequent write

W $\rightarrow$ **W** write must complete before subsequent write

As will be discussed in the next sections, different components, from the compiler to the processors to the memory system, can affect these orderings. In general, when analyzing source code, a change made by any of them is equivalent to any other such that they can be referred to and described as simple reorderings at the source level.

7.2.1 Sequential consistency (SC)

A *sequentially consistent* memory model (Lamport 1979) maintains all four memory operation orderings — the *program order* of loads and stores. In a sequentially consistent model, the following program (where `a` and `b` are global variables initially set to `0`) will output either `01` or `11`, but never `00` or `10`.

```
// thread 0          // thread 1
a = 1;               b = 1;
printf("%d", b);     printf("%d", a);
```

This can be demonstrated by exhaustively listing all the possible ways in which the four statements can be interleaved (figure 7.2). To simplify the notation, we use `a = b = 1` to denote

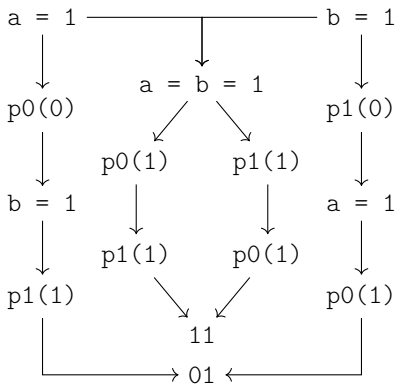


Figure 7.2: Sequentially consistent execution graph

```
bool flag0 = false;

void t0(void) {
    flag0 = true;
    if(flag1) {
        // contended
    } else {
        // critical section
        flag0 = false;
    }
}
```

```
bool flag1 = false;

void t1(void) {
    flag1 = true;
    if(flag0) {
        // contended
    } else {
        // critical section
        flag0 = false;
    }
}
```

Figure 7.3: Simplified Dekker's/Peterson's algorithm

the state where both stores happened before the loads, and `p0`/`p1` as shorthand for the `printf` in each thread.

This graph shows all four possible execution paths (with only two being externally distinguishable), which are much less than the $4! = 24$ permutations of the four initial statements. The strict sequentially consistent model establishes *happens before* relations (Lamport 1978) between every operation in a single thread, which can be seen in the graph as there are no edges that go from an operation to another that precedes it in the same thread. Thus, there are only two options: either one thread executes entirely before the other (in which case the output is `01`) or both stores happen followed by both loads (in which case the output is `11`).

Far from trivial, figure 7.3 shows that the same reasoning can be applied to solve a very important concurrent problem: mutual exclusion[2]. The loads and stores in this program are analogous to the ones in the previous execution graph. They can be analyzed in terms of the outputs described before:

- `00`: both processes see the flags as not set and enter the critical section. Impossible by the rules of sequential consistency.
- `01`: a process sees one of the flags unset and enters the critical section. The other process sees the other flag set and does not. This is the desired outcome.

[2]The code in this listing is a simplification of *Dekker's* and *Peterson's* algorithms. It is only concerned with mutual exclusion and is enough to demonstrate the effects of memory consistency. Those algorithms also handle contention and more than two processes and avoid starvation.

- `10`: a process sees one of the flags set without the other subsequently seeing the other flag set. Harmless, but also impossible in a sequentially consistent model.

- `11`: both processes observe the flags as set. None enters the critical section.

From these rules, it can be determined that mutual exclusion is guaranteed under the rules of the sequential consistency model. Sequential consistency is an intuitive and convenient model as it closely follows source code order. For this reason, it is the preferred model of many programming languages (*sequentially consistent for data-race-free programs*, SC-DRF). Maintaining sequential consistency, however, effectively serializes computation, negating all benefits of instruction- and thread-level parallelism. For this reason, no modern processor is fully sequentially consistent, as the synchronization cost would be prohibitive.

Most systems offer one of the many *relaxed memory consistency* models, allowing certain orderings to be violated. Each relaxation gives the hardware more freedom in how it can process instructions and service memory requests. A model that is more relaxed than sequential consistency commonly offers *barrier* (or *fence*) instructions to effectively reinstate sequential consistency at particular points in a program. A barrier forces some or all preceding memory operations to complete before some or all subsequent memory operations can start. These instructions are added only to the critical areas where inter-processor communication and synchronization is required, allowing the rest of the program to benefit from the many aforementioned hardware optimizations.

7.2.2 Total Store Ordering (TSO)

Relaxing the W $\rightarrow$ R ordering allows an independent read to be issued before a write completes. A store often must wait for an acknowledgment message from another CPU (v. 7.3, *Cache coherence*), which can take up to hundreds of cycles, so this can eliminate long stalls in the pipeline. It is a crucial transformation as loads usually occur at a much higher rate and are much less expensive to perform than stores, which require processor synchronization and flushing of caches to main memory. It is often the most important and common relaxation in a memory model.

A *write buffer* (or *store buffer*) in each processor can be used to allow reads to proceed while prior writes are executed. Stores are placed in the buffer and subsequent loads are executed immediately. Further writes are enqueued on the write buffer (i.e. there is no W $\rightarrow$ W reordering). From the point of view of the program, several instructions are now executed in parallel[3]. This is a different type of parallelism than separate threads being executed by multiple cores. In this scenario, called *instruction-level parallelism* (ILP), several instructions in a single instruction stream are executed in parallel.

The *x86/64* architecture and its *Total Store Ordering* model are an example of this relaxation. Synchronization instructions are required for instruction ordering not guaranteed by the model:

`lfence` waits for all loads to complete

`sfence` waits for all stores to complete

`mfence` waits for all operations to complete

[3]Of course, as described in the introduction of this section, the behavior described in this section is not directly visible to a well-behaved program. Nonetheless, they are indirectly observable. The program can, for example, measure that the execution of a sequence of instructions was done in less cycles than their combined cycle count. This is one reason why performance analysis and prediction in modern processors is difficult.

Even this single relaxation in the ordering of memory operations has profound consequences. Consider for example how the delay in the propagation of writes due to a store buffer affects the mutual exclusion example in the previous section. Barrier instructions now would have to be inserted to reinstate the sequentially consistent model to preserve the original order of operations. In general, any code in the form:

```
// thread0   // thread1
a = 1;       if(b)
b = 1;           assert(a);
```

will not work as expected in the presence of a store buffer. The reason for this is `b` might be already in the CPU's cache while `a` is not. The difference in propagation of stores could cause the other CPU to see them in the reverse order. The general solution is to add a memory barrier:

```
// thread0      // thread1
a = 1;          if(b)
_mm_mfence();       assert(a);
b = 1;
```

Upon executing the barrier instruction, the CPU will be prevented from transmitting further stores until all previous stores have been acknowledged and completed. This ensures `thread1` does not see the store to `b` before the store to `a`, thus preserving program order. It is not, however, quite sufficient to ensure the correctness of the program. The reason for this is another component associated with the store buffer, the *invalidate queue*.

A CPU which receives a request to remove a line from its cache can take a long time to respond if the cache is busy (e.g. when several *invalidate* messages arrive in a short period, as can happen immediately following a memory barrier if a CPU is executing many stores). The addition of an invalidate queue is based on the fact that responding to the request does not have to wait for the actual removal from the cache: as long as the CPU maintains internal consistency, it can respond immediately and place the invalidation in the queue, to be executed at a later time.

In the previous example, `thread1`'s immediate invalidation acknowledgement of `a` can cause `thread0` to progress past its memory barrier and execute the store to `b`. If `thread1` then receives the new value of `b` before processing the invalidation of `a`, it could still perceive the two stores in the opposite order. Therefore, a *second* memory barrier is needed:

```
// thread0      // thread1
a = 1;          if(b) {
_mm_mfence();       _mm_mfence();
b = 1;              assert(a);
                }
```

Note, however, that only the store buffer is of concern in `thread0` (the writer) and only the invalidate queue is of concern in `thread1` (the reader). There is no reason to affect the other component in each of the threads. For cases like this, which are very common, certain architectures provide specialized barrier instructions:

```
// thread0      // thread1
a = 1;          if(b) {
_mm_sfence();       _mm_lfence();
b = 1;              assert(a);
                }
```

The *store* and *load* fences will (roughly speaking) only operate on the store buffer and invalidate queue, respectively, allowing the other component to operate without restrictions.

7.2.3 Processor Consistency (PC)

Relaxing the W → W ordering allows various transformations of memory stores with respect to each other. The *SPARC* architecture is an example of this relaxation which allows write coalescing in the write buffer when they share the same cache line, saving memory bandwidth due to the reduced number of stores, or complete reordering of memory stores. Its *Processor Consistency* model further differs from TSO in that *any* processor can read the new value in a memory location before the store is observed by all processors, while in TSO reads by other processors cannot return the new value until the store is observed by *all* processors.

The simple program below demonstrates the difference between these two models. Under TSO, thread 2 cannot observe the write to `b` before the write to `a`. Under PC, however, there is no dependency between the two stores, so thread 2 may see the write to `b` before the write to `a`. Note that even the addition of a memory barrier in `thread1` would not guarantee correctness, since barriers only affect a given CPU's view of memory, i.e. there is no transitivity relation across CPUs.

```
// thread 0   // thread 1   // thread 2
a = 1;        while(!a);    while(!b);
              b = 1;        printf("%d", a);
```

Partial Store Ordering (PSO) is a more extreme relaxation of the same ordering, which allows complete reordering of memory stores. The following program would not match sequential consistency in this model, as there is no guaranteed order for the two stores in thread 0:

```
// thread 0   // thread 1
a = 1;        while(!flag);
flag = 1;     printf("%d", a);
```

7.2.4 Weak Ordering (WO), Release Consistency, (RC)

Weak ordering memory models[4]allow all four types of operation reordering. *Weak Ordering* and *Release Consistency* are examples of this type of model, one based on fences and the other on acquire/release semantics. *ARM* and *PowerPC* are examples of architectures with very relaxed consistency models.

7.3 Cache coherence

This section combines the material in sections 1.4.2 (*Cache*) and 7.2 (*Memory consistency*) to describe the protocols used by CPUs to keep all memory caches synchronized. Protocols used in modern CPUs are complex, but the basic *MESI* protocol can be used to describe the most fundamental aspects. It is named after its four states:

Modified lines are present only in this CPU's cache and have had their value changed. These lines are owned by the CPU and it must either write it back to main memory or transfer ownership to another cache at some point.

Exclusive lines are present only in this CPU's cache and contain the same value as the one in memory, but the CPU plans to modify its value in the near future (e.g. it is in the middle of a read-modify-write operation). These lines are also owned by the CPU but can be discarded at any time.

[4]"Weak ordering" is sometimes used *lato sensu* to refer to all models which are more relaxed than sequential consistency. The term is always used *stricto sensu* here.

Shared lines are present in this CPU's cache and potentially in others, and have not been modified. If it needs to be modified, the other CPUs have to be consulted.

Invalid lines are not present in the cache and are immediate targets for replacement by new lines added to the cache.

Each line in the cache has this two-bit state together with the address information and data. CPUs transition lines between states when necessary to effect operations by exchanging messages with other CPUs. Multicore processors are message-passing systems at their core. In systems where all CPUs are interconnected, the messages are:

Read requests the contents of a memory address.

Read Response contains the data requested by a *read* message, supplied either by the memory system or by a CPU which holds the line in the *modified* state.

Invalidate requests that a given address be removed from the caches.

Invalidate Acknowledge responds to an *invalidate* message.

Read Invalidate a combination of the *read* and *invalidate* messages. Requires a *read response* and a set of *invalidate acknowledge* responses.

Writeback writes data from a *modified* line to a memory address.

Lines can transition from one of the states to any of the other three, depending on the operations being performed by the CPUs:

M → **E** a CPU issues a *writeback* message to write the contents of a line to memory while retaining ownership of it.

E → **M** an already-owned line is modified, no messages are required.

M → **I** a CPU receives a *read invalidate* message for one of its *modified* lines. It removes the line from its cache and sends both a *read response* message with the data and an *invalidate acknowledge* message.

I → **M** a CPU issues a RMW operation on a line that is not in its cache. It sends a *read invalidate* message and receives the data in a *read response* message, along with a set of *invalidate acknowledge* messages, at which point it can complete the transition.

S → **M** a CPU issues a RMW operation on a line that is in its cache. It sends an *invalidate* message and receives a set of *invalidate acknowledge* messages, at which point it can complete the transition.

M → **S** a CPU receives a *read* message for one of its *modified* lines. It sends a *read response* with the data and possibly writes the line back to memory.

E → **S** a CPU receives a *read* message for one of its *modified* lines. It sends a *read response* with the data.

S → **E** a CPU will soon modify a line that is in its cache. It sends an *invalidate* message and receives a set of *invalidate acknowledge* messages, at which point it can complete the transition.

E → I a CPU receives a *read invalidate* message for one of its *exclusive* lines. It removes the line from its cache and sends both a *read response* message with the data and an *invalidate acknowledge* message.

I → E a CPU issues a store to a line not in its cache. It sends a *read invalidate* message and receives the data in a *read response* message, along with a set of *invalidate acknowledge* messages, at which point it completes the store and transitions to *modified*.

I → S a CPU loads a line not in its cache. It sends a *read* message and receives a *read response*.

S → I a CPU receives an *invalidate* message from another CPU doing a store to a *shared* line. It responds with an *invalidate acknowledge*.

7.4 Atomic operations

Section 7.2 (*Memory consistency*) described several memory consistency models adopted by different architectures. It also briefly described how weaker models provide specialized instructions to selectively and temporarily impose stronger guarantees. This section will present these operations, collectively called *atomic operations*, in more detail.

7.4.1 Basic properties

The term "atomic" is commonly (and often ambiguously) used to refer to several different aspects of the underlying machine concepts and operations, each explained in more detail in the following sections.

`volatile`

`volatile` type qualifiers act as compiler barriers, inhibiting most optimizations. "Volatile" is the name of a type qualifier (similar to `const`) used in C/++ to indicate that a variable may be modified by processes external to the thread of execution presented to the compiler, such as external hardware or other operating system threads[5]. They prevent the compiler from reordering loads and stores and from performing optimizations such as read/write coalescing.

In some architectures, this is sufficient to implement relaxed atomic operations (see below) for common types[6]. For this reason, it is found often in platform-specific code which predates the C/++ 11 memory model. When available, the atomic types introduced by that model are preferred, as they offer more control over which optimizations the compiler is allowed to perform. Code which uses `volatile` also often has to resort to more expensive, general fence instructions instead of the specific acquire/release operations available in some architectures.

Atomic loads/stores

These provide safe reads and writes by multiple threads. These are special (depending on the architecture) load/store operations which are guaranteed to be performed indivisibly. An external

[5] `volatile` can have different meanings in other languages and even compilers. In Java, it means sequential consistency (similar to operations on `std::atomic` types in C++ without explicit memory ordering). Microsoft's compiler adds non-standard acquire/release semantics to loads/stores (unconditionally in older versions, now, by default).

[6] E.g. the `READ_ONCE` and `WRITE_ONCE` macros in the Linux kernel, which have relaxed ordering but guaranteed atomicity, are essentially `volatile` loads/stores. See also `process/volatile-considered-harmful.rst` in the kernel documentation.

observer will either not see them, or see their result in its entirety: no partial result is allowed. In some architectures, such as x86, unadorned loads and stores up to a given size and properly aligned behave in this way. In other cases, specialized instructions or external synchronization may be required to avoid torn loads/stores.

In languages such as C/++ which provide low-level control of the memory ordering of atomic operations, regular atomic loads/stores that *do not* impose any ordering (i.e. are used just for their atomicity property) are called *relaxed* atomic operations. While inter-thread synchronization requires ordering constraints, relaxed atomic operations are useful in cases where no synchronization is needed or it is already provided via other means.

Acquire/release semantics

The main mechanism for memory ordering with atomic instructions in programming languages are operations with *acquire/release semantics*, which are counterparts that constrain the reordering of operations and synchronize memory accesses across threads.

Release semantics prevent reordering of operations preceding the atomic operation with itself. Other processors are guaranteed to see the result of the preceding operations before succeeding operations.

Acquire semantics prevent reordering of operations succeeding the atomic operation with itself. Other processors are guaranteed to see the result of the succeeding operations before preceding operations.

Note that each of these guarantees is unidirectional: operations can be moved down or up *inside* a critical section. Normally, this is counterproductive since it increases contention but may be beneficial when combined with other optimizations.

Acquire and release semantics are an important building block of concurrent algorithms because of their transitivity. This means they can be a hidden implementation detail of certain operations that in turn can be thought of as themselves having acquire/release semantics. Algorithms can then be built using those operations and reasoned about at a higher level.

The most common example is starting and joining threads. A function such as `pthread_create` has release semantics, meaning all memory accesses are guaranteed to be seen by the thread that is being started. Equivalently, `pthread_join` has acquire semantics, meaning all memory accesses of the thread that just finished are guaranteed to be seen by the waiting thread. This means algorithms where producers and consumers are separated by this type of fork-join boundary can often operate with no locks or even atomic operations at all: the implicit acquire/release semantics of starting and joining the worker threads is enough to guarantee safe access to shared data.

A lock or mutex (v. section 7.5.1, *Mutex*) is another common example: releasing a lock is done with release semantics so that operations in the critical section are guaranteed to be seen before the release, acquiring a lock is done with acquire semantics so that the acquisition is guaranteed to be seen before the operations in the critical section.

Fence/barrier instructions

These are machine instructions which establish memory ordering guarantees between different threads operating on the same data. They are the low-level mechanism that enables acquire/release semantics.

Some programming languages (such as C/++) provide fence operations in addition to the regular atomic types. Acquire and release fences are similar to atomic operations with acquire

and release semantics and provide similar guarantees. The one difference between them is an atomic operation is tied to the variable it is invoked on, so it prevents reorderings in relation to *itself*; a fence has no attached variable, so it prevents reorderings in relation to *all operations that precede or succeed it*, for acquire and release fences, respectively.

Read-modify-write

These operations are usually what is meant by "atomic operations" stricto sensu. These combine atomic loads/stores and fence/barrier instructions to modify shared data safely in multiple threads.

7.4.2 Optimization

Single-threaded programs give the compiler and the hardware many opportunities to transform the operations as listed in source code to better exploit machine capabilities and achieve better performance. It is worth analyzing these sorts of optimizations for a better understanding of why they have to be suppressed for the correct functioning of multi-threaded. They will be described from the point of view of the compiler, but for the most part are equivalent — at the source-code level — to transformations done by other components at the hardware level. Consult section 7.2 (*Memory consistency*) for a detailed description of the fundamentals that make these transformations legal under traditional memory models.

A compiler which encounters a series of redundant loads from a variable can decide to coalesce them to a single load:

```
x = g;
x = g;   →   x = g;
x = g;
```

This is because, in the absence of data races (which are expressively forbidden by language memory models), there are no legal ways in which the value of g could change between each load. Similarly, multiple redundant stores to the same variable can be coalesced to a single store of the final value. These optimizations can affect control structures as well:

```
while(g)          →      if(tmp = g)
    use(g);                  for(;;)
                                 use(g);
```

All of these cases are legal transformations because single-threaded behavior is maintained. They are catastrophic in multi-threaded code, however, where shared data accesses are used for inter-thread communication. Reorderings of this type must be prevented in some cases for an algorithm to function correctly. Conversely, it is desirable to do this *only when necessary*. These optimizations happen for a good reason, and excessive suppression can result in inefficient programs. For example, the compiler might decide to make the following transformation:

```
while(tmp = g)   →   while(g)
    use(tmp);            use(g);
```

This may happen due to register exhaustion; forbidding such transformation (e.g. by making g `volatile`) will result in `tmp` being spilled on the stack instead of the double load from g. Also note that, although there was no code surrounding the relevant operations in these examples, optimizations of operations involving other variables may also be affected.

Counter-intuitively, stores can also be added by the compiler, such as the following example, where a branch is removed with the addition of a second store:

```
if(i)                          g = 42;
    g = i;       →             if(i)
else                               g = i;
    g = 42;
```

7.4.3 Wait-free queue

One of the fundamental constructs in multi-threaded code is the FIFO queue. Many problems are structure in some form of producer-consumer relation, and a queue is often a good mechanism for communication between them. This section will demonstrate the use of atomic operations in the construction of concurrent queues of varying complexity.

Single-threaded

For illustrative purposes, we start with an extremely constrained implementation (listing 7.1), which 1) is not used concurrently, 2) only stores `int`s, 3) has a fixed size, 4) is either read- or write-only, and 5) can be read from / written to only once

```
struct queue {
    unsigned i;
    int v[N];
};

void push(struct queue *q, int i) {
    q->v[q->i++] = i;
}

void pop(struct queue *q, int *i) {
    *i = q->v[q->i++];
}

bool is_full(const struct queue *q) {
    return q->i == N;
}
```

Listing 7.1: Single-threaded queue

All these restrictions are imposed so that this first implementation can be almost non-existent and serve as a base for the incremental addition of concurrent facilities. Still, there are many scenarios in which a data buffer is read and/or written iteratively but received and/or sent in its entirety where this data structure is already useful. As mentioned in the description of acquire/release semantics in section 7.4.1 (*Basic properties*), the implicit guarantees of starting and joining threads is often enough to properly synchronize memory accesses in these cases.

Other than the buffer where the data are stored, the only other member is an index into this buffer, the read/write pointer. In both modes, checking if the queue is full — i.e. whether values can still be pushed/popped — is simple, since this is a single-threaded queue. In a read-only queue the only other operation is `pop`, which loads the current value and advances the pointer. Conversely, in a write-only queue the only other operation is `push`, which stores the value and advances the pointer.

Read-/write-only

The first concurrent application we will examine is multiple producers or multiple consumers. A queue is still read-/write-only, but multiple threads are now allowed to populate the buffer in parallel. This presents a problem as the expression `q->i++`, even though it is a single statement in C, is composed of multiple operations. Of relevance here is how the queue index is incremented in each function:

```
push:                                   mov 4[rdi+rax*4], esi
    mov eax, [rdi]                      ret
    lea edx, 1[rax]
    mov [rdi], edx                  pop:
```

```
mov eax, [rdi]
lea edx, 1[rax]
mov eax, 4[rdi+rax*4]
mov [rdi], edx
mov [rsi], eax
ret
```

```
push:
        ldr r3, [r0]
        add r2, r3, #1
        add r3, r0, r3, lsl #2
        str r2, [r0]
        str r1, [r3, #4]
        bx  lr
pop:
        ldr r3, [r0]
        add r2, r0, r3, lsl #2
        ldr r2, [r2, #4]
        add r3, r3, #1
        str r3, [r0]
        str r2, [r1]
        bx  lr
```

Incrementing the value of the index in memory involves separate load, increment, and store instructions, seen at the beginning of the x86 version of `push` in the listing above[7]. Since multiple threads may execute this sequence of instructions and could be interleaved in any order, there is the potential of data races and incorrect results (consider the case where threads are suspended before they can execute the third instruction and later resumed). For this queue to operate correctly, the increment of the index must be a serialized, atomic operation.

Naturally, one implementation possibility is to wrap operations with a mutex (v. section 7.5.1, *Mutex*), as shown in listing 7.2. This guarantees serialization and may be appropriate under low contention (v. Preshing 2011). In high-throughput queues, however, the system calls required to suspend and wake threads can become a limiting factor.

```
struct queue {
    mtx_t mtx;
    unsigned i;
    int v[N];
};

void queue_init(struct queue *q) {
    mtx_init(&q->mtx, mtx_plain);
}

void push(struct queue *q, int i) {
    mtx_lock(&q->mtx);
    q->v[q->i++] = i;
    mtx_unlock(&q->mtx);
}

void pop(struct queue *q, int *i) {
    mtx_lock(&q->mtx);
    *i = q->v[q->i++];
    mtx_unlock(&q->mtx);
}

bool is_full(struct queue *q) {
    mtx_lock(&q->mtx);
    const unsigned i = q->i;
    mtx_unlock(&q->mtx);
    return i == N;
}
```

Listing 7.2: Queue with mutex

Before analyzing an alternative implementation, we first look at a problem that follows from the introduction of multiple threads. In a single-threaded program, mere program order guarantees that consuming the data buffer is safe after the last `push` operation finishes or before the first `pop` starts. No such guarantee exists in a multi-threaded scenario, so memory ordering instructions are necessary. In these examples, unless otherwise noted, this ordering is assumed to be established by the operations which start the threads and wait for them to finish — i.e. that they have, respectively, acquire and release semantics.

Since there are only ever producers *or* consumers operating on the queue, it is enough to guarantee that the increments are serialized and the loads/stores are not reordered. Listing 7.3

[7] Note also that GCC in this optimized build (`gcc -O2`) chose to first store `q->i` *then* load/store the value `q->v` in both functions in both architectures. We will analyze this important fact in later implementations.

shows an implementation using a relaxed atomic increment operation, which has all of these properties.

```
struct queue {
    atomic_uint i;
    int v[N];
};

int *advance(struct queue *q) {
    return q->v + atomic_fetch_add_explicit(&q->i, 1, memory_order_relaxed);
}

void push(struct queue *q, int i) {
    *advance(q) = i;
}

void pop(struct queue *q, int *i) {
    *i = *advance(q);
}

bool is_full(const struct queue *q) {
    return atomic_load_explicit(&q->i, memory_order_relaxed) == N;
}
```

Listing 7.3: Queue with atomic operations

is_full

Lock-free algorithms are rarely completely general and thus must be built to fit particular usage and performance requirements. Different circumstances often require different algorithms, sometimes radically so. The current implementation has made `is_full` purely informational: it cannot be used to determine if the other operations are safe since the index may be moved between the check and the subsequent increment. It is assumed that there are other ways of guaranteeing that operations do not read/write outside the buffer (e.g. each thread is given an exact number of push/pop operations).

One alternative is to check the value returned by `atomic_fetch_add` before performing the load/store. The atomic increment has already ensured the thread has a unique position in the buffer by the time the check is done, so no race condition is introduced. The queue index is always incremented, however, so this is only safe if there are enough values between `N` and `UINT_MAX` (the maximum value for `q->i`) so that all threads can increment it once and see that it is now out of bounds. If not, the value will eventually wrap back to zero[8]and appear to be valid again.

Listing 7.4 shows a more robust implementation which eliminates these restrictions at the cost of more expensive atomic operations. It uses an *atomic compare-and-swap* (CAS) loop to ensure `q->i` is only assigned valid indices. A compare-exchange operation has three arguments (other than the memory ordering specification): the destination and the *expected* and *desired* values. It first compares the destination and expected values. If they are equal (i.e. the destination has not been modified by other threads since it was last loaded), it stores the desired value in

[8]N.b.: `q->i` is an `unsigned` value, so this is defined behavior. See also the discussion below of the ABA problem introduced by reusing index values.

the destination. If they are not, the expected value is updated with the current value of the destination. All of these steps happen as a single atomic operation.

```
int *advance(struct queue *q) {
    const memory_order rlx = memory_order_relaxed;
    for(unsigned i = atomic_load_explicit(&q->i, rlx); i != N;)
        if(atomic_compare_exchange_weak_explicit(&q->i, &i, i + 1, rlx, rlx))
            return q->v + i;
    return NULL;
}

bool push(struct queue *q, int i) {
    int *const p = advance(q);
    return p && (*p = i, true);
}

bool pop(struct queue *q, int *i) {
    const int *const p = advance(q);
    return p && (*i = *p, true);
}
```

Listing 7.4: Queue with atomic compare-and-swap

The queue index is first loaded using a relaxed atomic operation. If the value is ever observed to be at the end of the buffer, the `for` loop terminates and `NULL` is returned. In each iteration, we increment the value just loaded and attempt to store it, checking to see if it has not changed while it was being tested. This type of construction, where an atomic CAS is used to build an effectively atomic operation out of one or more non-atomic ones, is a very common pattern in lock-free algorithms.

Ring buffer

An alternative design is to let threads continuously operate on the queue even when there is no space left. Whenever the index reaches the end of the queue, it is reset to the beginning (conceptually, at least). This arrangement is usually called a *ring buffer*, since the wrapping of the index space can be conceptualized as a circle (*circular buffer* is also commonly used).

Push operations overwrite older values when there is no space left on the queue. This may be desirable if only the last N or less items are relevant, which is common, for example, in real-time systems. The pop equivalent is less common, but may be useful if threads consume a finite, pre-calculated set of data. When the buffer is exhausted, the index is simply reset to the beginning and threads start reusing old values, potentially in a different order.

Listings 7.5 and 7.6 show implementations of this pattern. The only change is to the `advance` function, since it is the one which defines the index space. `is_full` has no meaning in ring buffers, so it is not included. Listing 7.5 is implemented in a general way, using the integer modulus operation. Since this is a relatively expensive operation, a common choice is to limit the queue size to powers of 2. The remainder can then be found with a simple bitwise `and` operation, which is usually an order of magnitude faster. Listing 7.6 shows this implementation[9].

[9] V. section 1.1.1 for an explanation of the expression used in the `static_assert`.

```
int *advance(struct queue *q) {
    const unsigned i = atomic_fetch_add_explicit(
        &q->i, 1, memory_order_relaxed);
    return q->v + i % N;
}
```

Listing 7.5: Ring buffer

```
int *advance(struct queue *q) {
    const unsigned i = atomic_fetch_add_explicit(
        &q->i, 1, memory_order_relaxed);
    static_assert(!(N & (N - 1)));
    return q->v + (i & (N - 1));
}
```

Listing 7.6: Ring buffer with bitwise and

ABA

This is our first encounter with the infamous *ABA problem*, so called after the case where a thread observes value `A` and is suspended, the value is changed from `A` to `B` then back to `A`, and the original thread is resumed. Even though there was no change from the perspective of the waking thread, the two values of `A` may not represent the same state. The canonical example is that of a pointer `A` to an object which is deallocated and whose memory block is subsequently reused[10]. A thread using the memory address contained in `A` as an identifier will perceive no change, even though the object to which it points is now different.

Our ring buffer suffers from a different type of ABA problem, stemming from the fact that there is an interval between the calculation of the new index by `advance` and the store in `push`. All previous examples relied on the fact that indices are not reused. This is also why only the access to the queue index is done using atomic operations: loads and stores to a particular buffer position are always done by a single thread. This is not true in a ring buffer, but the serialization of the atomic increment operation is used to ensure that each thread issues stores to a memory location it has exclusive access to.

However, it is possible that a thread could be suspended right after calculating its index and sleep while other threads go through a full rotation of the ring. When the thread is awakened, it could attempt to write to the memory location it originally calculated at the same time as another thread which just happened to be assigned the same index, now a full rotation ahead. Whether this is a problem in reality is a function of the queue size and frequency of operations. The larger the queue size the less likely a thread will be suspended for long enough for the same index to be reused. There are only a few machine instructions involved, so the window for thread preemption is small:

```
push:
    mov       eax, 1
    lock xadd DWORD PTR [rdi], eax
    and       eax, 1023
    mov       4[rdi+rax*4], esi
    ret
```

```
push:
.L5:
    ldrex r3, [r0]
    add   r2, r3, #1
    strex ip, r2, [r0]
    cmp   ip, #0
```

[10] It is common for memory allocators to use some kind of MRU policy since it increases the chance of maintaining blocks in cached memory.

```
bne   .L5
ubfx  r3, r3, #0, #10
add   r0, r0, r3, lsl #2
str   r1, [r0, #4]
bx    lr
```

Single-producer, single-consumer

The queue implementations examined so far required no thread synchronization: since only producers *or* consumers were involved, it was enough to ensure that the operations on shared memory were atomic (i.e. indivisible). The implicit serialization of the writes to the queue index was all that was needed to arbitrate access to the buffer.

We now consider a single-producer, single-consumer queue (listing 7.7). There are now two indices: `r` and `w`, the read and write pointers. They are the equivalent of the previous single index `i` for the consumer and the producer, respectively, but allow them to share access to the buffer and operate at different frequencies. The write pointer is always at or ahead of the read pointer. We will initially return to a single-use queue, so in this example both pointers advance only once to the end of the buffer. We will later reconsider ring buffers.

```
struct queue {
    volatile unsigned w;
    unsigned r;
    int v[N];
};

bool push(struct queue *q, int i) {
    const unsigned w = q->w;
    if(w == N)
        return false;
    q->v[w] = i;
    q->w = w + 1;
    return true;
}

bool pop(struct queue *q, int *i) {
    const unsigned w = q->w, r = q->r;
    if(r == w)
        return false;
    *i = q->v[r];
    q->r = r + 1;
    return true;
}
```

Listing 7.7: (Incorrect) SP/SC queue

The implementation as shown is incorrect because relaxed atomic operations are no longer sufficient: the queue now requires memory ordering operations to synchronize the producer and the consumer. To understand why, we must analyze the sequence of operations performed by each in `push` and `pop`:

- The read pointer is only ever modified by the (single) consumer in `pop`, so it is a regular `unsigned` variable.
- The write pointer is similarly only ever modified by the (single) producer in `push`, but this variable is also read by the consumer. We signal this by the use of `volatile`[11].
- To pop a value from the buffer, the consumer verifies that the producer has made data available, loads it, and advances the read pointer.
- To push a value to the buffer, the producer verifies that there is still space remaining (this is now a valid check as there is a single producer), stores it, and advances the write pointer.
- Loads and stores to the indices and the buffer must happen in program order, otherwise the consumer could see stale or uninitialized data.

[11] Assuming loads/stores of `unsigned` values are atomic, relaxed atomic operations would be the portable mechanism.

Consider what can happen without memory ordering restrictions in a weakly-ordered architecture. The two stores in push could be effected by the hardware in the opposite order[12]:

```
q->v[w] = i;          →   q->w = w + 1;
q->w = w + 1;             q->v[w] = i;
```

If the reader side is executed between these two conceptual lines, it will read the updated write position and load stale or uninitialized memory. Similarly, the two loads in pop could see values stored at different points in time, each from a separate call to push:

```
const unsigned w = q->w, r = q->r;        *i = q->v[q->r];
// ...                                →   const unsigned w = q->w, r = q->r;
*i = q->v[q->r];                          // ...
```

Listing 7.8 shows a revised implementation that eliminates these problems using explicit memory ordering operations. The corresponding atomic load/store of q->w in push/pop guarantee thread synchronization. The fence operation with *acquire* semantics guarantees that all loads that succeed it see values stored after the corresponding store was executed[13]. The store and all operations preceding it are said to *happen before* the fence and all operations that succeed it. This relationship is denoted with the right arrow ($\rightarrow$) symbol: $l \rightarrow s$, load l happens before store s.

```
struct queue {
    atomic_uint w, r;
    int v[N];
};

bool queue_try_push(struct queue *q, int i) {
    const unsigned w = q->w;
    if(w >= N)
        return false;
    q->v[w] = i;
    atomic_store_explicit(&q->w, w + 1, memory_order_release);
    return true;
}

bool queue_try_pop(struct queue *q, int *i) {
    const unsigned w = atomic_load_explicit(&q->w, memory_order_relaxed);
    const unsigned r = q->r;
    if(r >= w)
        return false;
    atomic_thread_fence(memory_order_acquire);
    *i = q->v[r];
    q->r = r + 1;
    return true;
}
```

Listing 7.8: SP/SC queue

[12]The volatile type qualifier of q->w is only applied at the language/compiler level. It does not affect the types of loads/stores ultimately performed by the machine.

[13]The acquire operation is not needed in the case where there is no data to be read, so a combination of a relaxed load + fence is used. A single load/acquire operation would be equally valid, but with a potential efficiency loss. The load of q->w at the beginning of push can be non-atomic since it is only ever written to by the (single) producer.

This eliminates the first problem described above as the two instructions cannot be reordered[14]:

```
q->v[w] = i;
q->w = w + 1;   ⟶   const unsigned w = /*...*/;
                    *i = q->v[r];
```

That is, any time the reader side loads a value into its `w` local variable, all instructions following that load are guaranteed to see all stores that preceded the corresponding store. It will never see the store to `q->v[w]` on the writer side without also seeing the immediately-preceding store to `q->w`. The load operation synchronizes with the store, creating a barrier around which stores cannot be reordered.

It is important to note that this relationship exists only between two corresponding load/store operations. A `pop` operation that occurs in between a series of `push`es could see any of the possible values of `q->w`, depending on how the producer/consumer are interleaved. The only guarantee provided by memory ordering operations in this respect is that a subsequent load will not see previous values of `q->w` (or any other preceding store).

```
q->v[w] = i;
q->w = w + 1;
q->v[w] = i;
q->w = w + 1;   ⟶   const unsigned w = /*...*/;
q->v[w] = i;        *i = q->v[r];
q->w = w + 1;
q->v[w] = i;
q->w = w + 1;   ⟶   const unsigned w = /*...*/;
                    *i = q->v[r];
```

Similarly, this "happens before" relation is only a *strong partial order* (v. section 2.3.1, *Ordering*): it establishes no relation between two operations that happen before a third. In other words, two operations are concurrent if $\neg((a \to b) \vee (b \to a))$. It is, however, transitive: $(a \to b) \wedge (b \to c) \implies a \to c$. This can be used to build a causal chain of arbitrary length between successive operations in two or more threads, each of which is guaranteed to occur after all the operations that happen before it.

7.4.4 Code generation

This section will explore the concepts discussed so far in this chapter by analyzing the machine code generated for several architectures for the following C code[15]:

```
#include <stdatomic.h>

atomic_int x;
int load_rlx(void) { return atomic_load_explicit(&x, memory_order_relaxed); }
```

[14] In this and the following diagrams, the long arrow symbol (⟶) indicates the "happens before" relation described above between synchronized pairs of loads/stores in the code fragments immediately to its left and right.

[15] Machine code generated using:

- x86: `gcc -std=c11 -O2` (11.2)
- ARMv7: `clang -march=armv7 -std=c11 -O2` (11.0.1)
- ARMv8: `gcc -march=armv8-a -std=c11 -O2` (11.2)
- PPC: `gcc -std=c11 -O2` (9.2.1)

```
int load_acq(void) { return atomic_load_explicit(&x, memory_order_acquire); }
int load(void) { return atomic_load(&x); }
void store_rlx(int i) { atomic_store_explicit(&x, i, memory_order_relaxed); }
void store_rel(int i) { atomic_store_explicit(&x, i, memory_order_release); }
void store(int i) { atomic_store(&x, i); }
void xchg_w(int i0, int i1) { atomic_compare_exchange_weak(&x, &i0, i1); }
void xchg_s(int i0, int i1) { atomic_compare_exchange_strong(&x, &i0, i1); }
```

x86

The x86 architecture provides the very strict TSO model. The only reordering allowed is that loads can proceed while stores are being processed. In particular, loads are not reordered with any other loads, stores are not reordered with any other stores, and stores are not reordered with previous loads.

This means there is no difference between regular loads and load/acquire operations: they both translate to the `mov` instruction, and so all `load` and `store` functions, except for the sequentially consistent store, are simple `mov`s. The store/release and compare-exchange operations do need specialized instructions, `xchg` / `lock cmpxchg`, which act as a full barrier (so there is no difference between the weak and strong CAS).

```
load_rlx: # same for load_acq, load
    mov eax, DWORD PTR x[rip]
    ret
store_rlx: # same for store_rel
    mov DWORD PTR x[rip], edi
    ret
```

```
store:
    xchg edi, DWORD PTR x[rip]
    ret
xchg_w: # same for xchg_s
    mov eax, edi
    lock cmpxchg DWORD PTR x[rip], esi
    ret
```

ARM

When considering the ARM architecture, revision ARMv8-A and its successors have to be analyzed separately. That is because they introduced specialized instructions dedicated to implementing release consistency to conform to the C/++11 sequentially consistent atomic types:

```
load_rlx:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    ldr r0, [r3]
    bx lr
```

```
load_acq:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    lda r0, [r3]
    bx lr
```

```
load:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    lda r0, [r3]
    bx lr
```

```
store_rlx:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    str r0, [r3]
    bx lr
```

```
store_rel:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    stl r0, [r3]
    bx lr
```

```
store:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    stl r0, [r3]
    bx lr
```

Prior versions, such as ARMv7, require full memory barriers to implement that model:

```
load_rlx:
    movw r3, #:lower16:x
    movt r3, #:upper16:x

    ldr r0, [r3]

    bx lr
```

```
load_acq:
    movw r3, #:lower16:x
    movt r3, #:upper16:x

    ldr r0, [r3]
    dmb ish
    bx lr
```

```
load:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    dmb ish
    ldr r0, [r3]
    dmb ish
    bx lr
```

```
store_rlx:
    movw r3, #:lower16:x
    movt r3, #:upper16:x

    str r0, [r3]

    bx lr
```

```
store_rel:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    dmb ish
    str r0, [r3]

    bx lr
```

```
store:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    dmb ish
    str r0, [r3]
    dmb ish
    bx lr
```

Also contrary to x86, strong compare-exchange is implemented as a loop in machine code, meaning the weak version can be advantageous when it is already surrounded by another loop:

```
xchg_w:
    movw r3, #:lower16:x
    movt r3, #:upper16:x

    ldaex r2, [r3]
    cmp r2, r0

    stlexeq ip, r1, [r3]

    bx lr
```

```
xchg_s:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
.L11:
    ldaex r2, [r3]
    cmp r2, r0
    bxne lr
    stlex ip, r1, [r3]
    cmp ip, #0
    bxeq lr
    b .L11
```

Additionally, full memory barriers are required for earlier revisions as described above:

```
xchg_w:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    dmb ish

    ldrex r2, [r3]
    cmp r2, r0
    it eq
    strexeq ip, r1, [r3]



    dmb ish
    bx lr
```

```
xchg_s:
    movw r3, #:lower16:x
    movt r3, #:upper16:x
    dmb ish
.L11:
    ldrex r2, [r3]
    cmp r2, r0
    bne .L12
    strex ip, r1, [r3]
    cmp ip, #0
    bne .L11
.L12:
    dmb ish
    bx lr
```

PowerPC

PowerPC also has a relaxed memory model and, similarly to ARMv7, requires memory barriers to implement acquire/release operations.

```
load_rlx:
    addis 9,2,.LC0@toc@ha
    ld 9,.LC0@toc@l(9)

    lwz 3,0(9)



    extsw 3,3
    blr
```

```
load_acq:
    addis 9,2,.LC1@toc@ha
    ld 9,.LC1@toc@l(9)

    lwz 3,0(9)
    cmpw 0,3,3
    bne- 0,$+4
    isync
    extsw 3,3
    blr
```

```
load:
    addis 9,2,.LC2@toc@ha
    ld 9,.LC2@toc@l(9)
    sync
    lwz 3,0(9)
    cmpw 0,3,3
    bne- 0,$+4
    isync
    extsw 3,3
    blr
```

```
store_rlx:
    addis 9,2,.LC3@toc@ha
    ld 9,.LC3@toc@l(9)

    stw 3,0(9)
    blr
```

```
store_rel:
    addis 9,2,.LC4@toc@ha
    ld 9,.LC4@toc@l(9)
    lwsync
    stw 3,0(9)
    blr
```

```
store:
    addis 9,2,.LC5@toc@ha
    ld 9,.LC5@toc@l(9)
    sync
    stw 3,0(9)
    blr
```

```
xchg_w:
    addis 9,2,.LC5@toc@ha
    ld 10,.LC5@toc@l(9)
    sync

    lwarx 9,0,10
    cmpw 0,9,3
    bne 0,.L8
    stwcx. 4,0,10

.L8:
    isync
    blr
```

```
xchg_s:
    addis 9,2,.LC6@toc@ha
    ld 10,.LC6@toc@l(9)
    sync
.L10:
    lwarx 9,0,10
    cmpw 0,9,3
    bne 0,.L11
    stwcx. 4,0,10
    bne 0,.L10
.L11:
    isync
    blr
```

7.5 Semaphores

A semaphore is a concurrency primitive that limits access to a section of code. It can be conceptualized as a simple unsigned integer and associated operations to decrement and increment its value, traditionally called P and V[16].

P decrements the value of the semaphore. This operation can be performed as long as the value is greater than zero. A thread which attempts to decrement a semaphore with value less than or equal to zero is suspended. This operation is also often called *acquiring* or *locking* the semaphore.

V increments the value of the semaphore. If there are any suspended threads, one of them is awakened. This operation is also often called *releasing* or *unlocking* the semaphore.

From this description, we derive that a semaphore is implemented in terms of a counter, a list of suspended threads, and a lower-level primitive to synchronize access to them (figure 7.4. A thread performing a decrement first locks the semaphore, then checks the counter. If it is zero,

[16]The initials used by Edsger W. Dijkstra in the papers that introduced these concepts, thought to be Dutch words used in railway systems that roughly translate to “passing” (the noun) and “release”.

the thread adds itself to the wait list and is suspended; otherwise, it continues execution. A thread performing an increment locks the semaphore, increments the value, and pops one thread from the wait list (if it is not empty) and resumes it. In all cases, the thread releases the lock at the end of the operation or before being suspended.

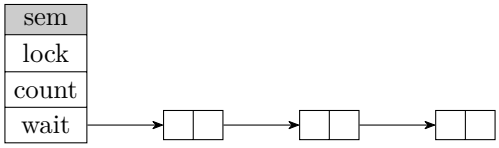


Figure 7.4: Semaphore

The region of code between the semaphore decrement/increment pair of calls must usually only be executed by a limited number of threads. The initial value of the semaphore and the semantics of the increment/decrement operations can be used to satisfy this constraint.

7.5.1 Mutex

Semaphores whose initial value is *one* are the most common. The region of code such semaphore protects is called a *critical section*, since the value of the semaphore guarantees that only a single thread can execute it at any point in time. The code in the critical section is therefore effectively *atomic*: the observable behavior is that all of its executions are serialized. A semaphore of this type is referred to as a *binary semaphores* or *mutex* (short for *mutual exclusion*).

An atomic boolean flag can be used to implement the simplest possible (binary spin-)lock (listing 7.9).

```
class mutex {
    using enum std::memory_order;
    std::atomic_flag f;
public:
    void lock(void) { while(this->f.test_and_set(acquire)); }
    void unlock(void) { this->f.clear(release); }
};
```

Listing 7.9: Simple lock using atomic operations

7.5.2 R/W semaphore

Both the P/V operations on a semaphore treat the calling thread indiscriminately. Often, however, not all scenarios require strict mutual exclusion. One of the most common is where an unlimited number of *readers* is allowed, but only one *writer* should be allowed to enter the critical section. A *read/write semaphore* is a concurrency primitive that satisfies these constraints. Reader threads can acquire the semaphore without contention as long as there are no writers. When a thread wishes to acquire the semaphore for writing, it blocks new readers from acquiring it and waits until all current readers are done. From then on, it has exclusive access to the semaphore.

This prioritization of writers means readers can be denied access for long periods of time (i.e. *reader starvation*). For this reason, R/W semaphores are best suited when reads are frequent and writes are fast and rare, but can result in significantly better performance in those cases.

7.5.3 `futex(2)`

A *futex* (*Fast User-space muTEX*) is a primitive in the Linux kernel used to build higher-level concurrency primitives, introduced in 2003 in the 2.6 stable series by Hubertus Franke, Matthew Kirkwood, Ingo Molnár, and Rusty Russell. The "fast" portion of the name refers to its user-space component, which can be as simple as a single atomic operation on an integer. The `futex(2)` system call handles the slow case of suspending threads when there the lock is contended. Futexes are fast because the cost of a system call can be completely avoided when there is no contention.[17]

The wait queue of suspended threads exists on the kernel side and is associated with the memory address supplied by user space. The `futex(2)` system call is a multiplexer (in the same manner as `ioctl(2)`) whose two main operations are `FUTEX_WAIT` and `FUTEX_WAKE`. `WAIT` adds the current thread to the wait queue (after ensuring — atomically — that the value being waited on has not changed), while `WAKE` resumes threads currently on the queue. Other operations are available as more specialized versions of `WAKE`:

- `FUTEX_CMP_REQUEUE`: wakes a specified number of threads and transfers any remaining ones to a second queue (avoids a *thundering herd* on wake)
- `FUTEX_WAKE_OP`: provides a more flexible waking comparison operation based on a number of available operations on a second value

Higher-level concurrency primitives are built using these two operations. One of the simplest is an event that can be signaled and waited on (extracted, slightly reformatted, from Drepper 2011, v. for further examples and discussion):

```
class event {
public:
    void signal() { futex_wake(&++this->val, INT_MAX); }
    void wait() { futex_wait(&this->val, this->val); }
private:
    int val = {};
};
```

7.5.4 Deadlock

Whenever a thread attempts to acquire a lock, it may put itself in a situation called a *deadlock*. This happens when the locking operation is blocking and results in a situation where no sequence of operations will ever unblock the thread and allow for forward progress. Two common cases of deadlock are when a thread attempts to acquire a lock it currently already holds, or when it needs to acquire more than one lock.

A *recursive lock* can be re-acquired even if a thread currently holds it. There is often a performance cost to this implementation, so regular locks are generally not recursive. Whether a thread which attempts to lock a non-recursive lock while holding it immediately deadlocks or receives an error is usually defined by the implementation, but the former case is more common.

Two general strategies exist for the case where multiple locks must be acquired. The simplest solution is to establish a hierarchy of locks and guarantee that they are always acquired in the same order by all threads. This prevents deadlocks since they are caused by two threads acquiring two different locks then attempting to acquire the one just acquired by the opposite thread. Another solution when a hierarchy cannot be easily established is to use the "speculative locking" operation commonly provided by implementations, where a thread can attempt to acquire a lock

[17] V. section 7.7 (*RCU*) for another example of optimizing for the uncontended case.

without blocking if it is already taken. Threads can use this operation when acquiring all locks after the first, releasing all of them if it fails and retrying the entire operation.

7.6 pthreads

7.6.1 Condition variable

Kerrisk 2010, 30.2

7.7 RCU

https://preshing.com/20160726/using-quiescent-states-to-reclaim-memory

7.8 Case studies

WIP

7.8.1 Double-checked locking

TODO

https://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11

```
std::atomic<C*> C::m_instance = {nullptr};

C *C::instance(void) {
    if(C::m_instance.load() == nullptr) {
        std::lock_guard lock{C::mtx_w};
        if(C::m_instance.load(std::memory_order_relaxed) == nullptr)
            C::m_instance.store(new C, std::memory_order_relaxed);
    }
    return C::m_instance.load();
}
```

7.8.2 Relaxed atomic operations

A counter which has no effect on memory ordering and is externally synchronized.

```
void thread(void) {
    for(;;) {
        // ...
        counter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main(void) {
    start_workers(thread);
    join_workers(thread);
    std::cout << counter.load(std::memory_order_relaxed) << '\n';
}
```

A stop signal for worker threads.

```
void thread(void) {
    while(!stop.load(std::memory_order_relaxed))
        // ...
}

int main(void) {
    start_workers(thread);
    stop.store(std::memory_order_release);
    join_workers(thread);
}
```

Incrementing a reference counter.

```
class ref {
    using enum std::memory_order;
    std::atomic<unsigned> c;
public:
    void inc(void) { this->c.fetch_add(1, relaxed); }
    bool dec(void) { return this->c.fetch_sub(1, acq_rel); }
};
```

7.8.3 Hash table

https://preshing.com/20130529/a-lock-free-linear-search
https://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table
https://preshing.com/20160201/new-concurrent-hash-maps-for-cpp
https://preshing.com/20160222/a-resizable-concurrent-map

7.8.4 Concurrent pipeline

The simplest form of concurrency is the concurrent pipeline. Tasks that initially are unnecessarily executed serially are moved to a separate thread. The main thread then simply dispatches work to the worker threads and possibly waits for them to complete at a later point in time.

Figure 7.5 shows a hypothetical graphics engine with a serial pipeline. At each frame, some processing of the data to be displayed happens, followed by calls to the graphics hardware. Once the data has been processed, displaying them is an independent operation, and so is the next frame's processing. Figure 7.6 shows the same process with a concurrent pipeline. A separate thread of execution receives a signal to start the rendering process, and the main thread is free to continue to the next frame. The engine and graphics tasks are now more packed, allowing more of them to be completed in the same amount of time.

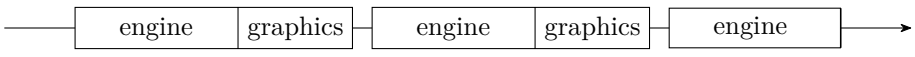


Figure 7.5: Serial pipeline

7.8.5 Dedicated thread

For tasks that may take more than one frame to complete and are not immediately necessary in the next, a dedicated thread can be used. This is a thread that remains idle until work is pushed to it, at which point it will process it, taking as much time as necessary, then signal back to the main thread that the work has been done. Figure 7.7 shows this process.

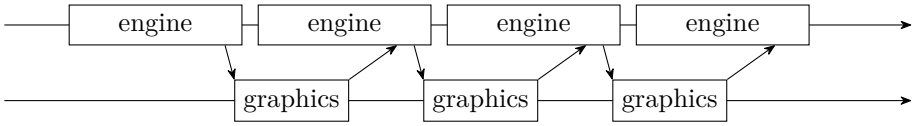


Figure 7.6: Concurrent pipeline

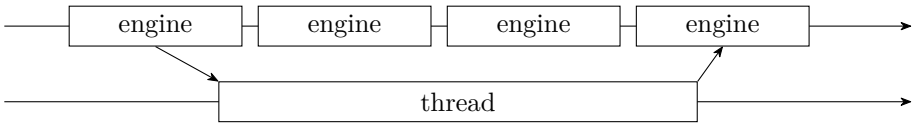


Figure 7.7: Dedicated thread

```
void thread(struct queue *q, struct event *e) {
    for(;;) {
        event_wait(e);
        for(struct task t = {0}; queue_try_pop(q, &t);)
            process_task(&t);
    }
}

void engine(void) {
    struct queue q = init_queue();
    struct event e = init_event();
    start_thread(thread, &q, &e);
    for(;;) {
        queue_push(&q, process_data());
        event_signal(&e);
    }
}
```

7.8.6 Task scheduler

```
void thread(struct queue *q, struct event *e) {
    for(;;) {
        event_wait(e);
        for(struct task t = 0; queue_try_pop(q, &t);)
            process_task(&t);
    }
}

void async(struct queue *q, struct event *e, struct task t) {
    queue_push(q, t);
    for(int i = 0; i < N_THREADS; ++i)
        event_signal(e + i);
}

void engine(void) {
    struct queue q = init_queue();
    struct event e[N_THREADS] = {0};
    for(int i = 0; i < N_THREADS; ++i) {
        e[i] = init_event();
        start_thread(thread, &q, e + i);
    }
```

```
        for(;;) {
            async(q, e, task0());
            async(q, e, task1());
            async(q, e, task2());
        }
    }
```

Chapter 8

Solutions to exercises

1.1.1.

The maximum number of bits for a given integer type (i.e. its bit width) is strictly less than the maximum value of that type[1]. However, it is significantly *less* than its maximum value in a sub-linear relationship. Recall that the maximum value of an unsigned type of width w is $2^w - 1$. Therefore, the relationship between the maximum value and the maximum number of bits is logarithmic.

The sized types mentioned in the C11 standard (*ISO/IEC 9899:2011* 2011) are in the range `uint8_t` to `uint64_t` (i.e. 8 to 64 bits wide), so the range of `uint8_t` is more than sufficient ($255 \gg 64$) to store the width of these types. There is no limit imposed on maximum width an implementation can provide, however, so a larger type would have to be used if widths greater than 255 exist.

1.1.2.

This solution uses a type alias implemented as `decltype` of the return value of a `consteval` function.

```
#include <bit>
#include <concepts>
#include <cstddef>
#include <cstdint>
#include <climits>

namespace detail {

template<std::size_t w>
consteval auto min_type_for_width_impl() {
    constexpr auto max = std::bit_width(w - 1);
    if constexpr(max < CHAR_BIT)
        return std::uint8_t{};
    else if constexpr(max < 2 * CHAR_BIT)
        return std::uint16_t{};
    else if constexpr(max < 4 * CHAR_BIT)
        return std::uint32_t{};
    else if constexpr(max < 8 * CHAR_BIT)
        return std::uint64_t{};
}
```

[1] Ignoring the case of `CHAR_BIT == 1`, where the size and width of `char` would be the same — i.e. `1`.

```
}

template<std::unsigned_integral T>
using min_type_for_width = decltype(
    detail::min_type_for_width_impl<sizeof(T) * CHAR_BIT>());
```

Listing 8.1: `min_type_for_width`

We can then assert that `uint8_t` can be used for all standard sized types:

```
template<std::unsigned_integral T, std::unsigned_integral WT>
using check = std::is_same<min_type_for_width<T>, WT>;

static_assert(check<std::uint8_t, std::uint8_t>());
static_assert(check<std::uint16_t, std::uint8_t>());
static_assert(check<std::uint32_t, std::uint8_t>());
static_assert(check<std::uint64_t, std::uint8_t>());
```

Listing 8.2: Tests for `min_type_for_width`

Relaxing the type constraint allows us to check with fictitious integer types:

```
#include <array>

template<typename T, std::unsigned_integral WT>
using check = std::is_same<min_type_for_width<T>, WT>;

template<std::size_t W>
using fake_uint = std::array<std::uint8_t, W / CHAR_BIT>;

static_assert(check<fake_uint<1ul << 7>, std::uint8_t>());
static_assert(check<fake_uint<1ul << 8>, std::uint16_t>());
static_assert(check<fake_uint<1ul << 15>, std::uint16_t>());
static_assert(check<fake_uint<1ul << 16>, std::uint32_t>());
static_assert(check<fake_uint<1ul << 31>, std::uint32_t>());
static_assert(check<fake_uint<1ul << 32>, std::uint64_t>());
```

Listing 8.3: Extended tests for `min_type_for_width`

1.2.

A chess board consists of an 8x8 grid with two identical sets, except for the color, of sixteen pieces each:

0. 8 pawns (p)
1. 2 rooks (r)
2. 2 knights (n)
3. 2 bishops (b)
4. 1 queen (q)
5. 1 king (k)

```
  A   B   C   D   E   F   G   H
+-------------------------------+
8 | r |:n:| b |:q:| k |:b:| n |:r:|
  |---+---+---+---+---+---+---+---|
7 |:p:| p |:p:| p |:p:| p |:p:| p |
  |---+---+---+---+---+---+---+---|
6 |   |:::|   |:::|   |:::|   |:::|
  |---+---+---+---+---+---+---+---|
5 |:::|   |:::|   |:::|   |:::|   |
  |---+---+---+---+---+---+---+---|
4 |   |:::|   |:::|   |:::|   |:::|
  |---+---+---+---+---+---+---+---|
3 |:::|   |:::|   |:::|   |:::|   |
  |---+---+---+---+---+---+---+---|
2 | P |:P:| P |:P:| P |:P:| P |:P:|
  |---+---+---+---+---+---+---+---|
1 |:R:| N |:B:| Q |:K:| B |:N:| R |
  +-------------------------------+
```

Listing 8.4: Chess board

Usually the colors of sets are white and black. In addition, every piece of the same type and color is interchangeable: all white pawns are equal, etc. Figure 8.4 shows a representation of the initial disposition of the pieces on the board using standard chess notation, with the pieces represented each by one character as defined in the list above. White pieces are shown in capital letters and black positions on the board have the shading character sequence `:::`.

An initial design for the storage is as a one- or two-dimensional array of total size 64, with each element corresponding to one position in a predefined order (e.g. row-major in board order). Any given position can be empty or contain one of the six types of pieces of one of the two colors. The empty position and the six piece types fit into three bits with one unused value, a fourth bit can hold the color.

All the information for a position is encoded in 4 bits (half a byte). Thus, the total space required for this representation is $8 \times 8 \times 4 = 256$ bits $= 32$ bytes. Listing 8.5 shows the implementation.

Storing information for empty positions is wasteful, specially as the game progresses and the board becomes more sparse. Even so, storing only the position data along with an index does not reduce the storage requirement. Storing the piece and color still takes four bits (now with a second unused value), and six bits are required for an index into the array ($\log_2 64 = 6$), yielding a total of $32 \times 10 = 320$ bits $= 40$ bytes for the initial board with all pieces for maximally packed (but now unaligned) data.

```
enum {
    EMPTY, PAWN, ROOK, KNIGHT,
    BISHOP, QUEEN, KING,
    POS_BITS = 4,
    POS_MASK = (1 << POS_BITS) - 1,
    PIECE_BITS = 3,
    PIECE_MASK = (1 << PIECE_BITS) - 1,
    COLOR_MASK = (1 << PIECE_BITS),
    WHITE = 0, BLACK = 1,
};

typedef uint8_t position;
struct board { position v[32]; };

position position0(uint8_t i) {
    return i & POS_MASK;
}

position position1(uint8_t i) {
    return i >> POS_BITS;
}

bool is_empty(position p) {
    return !p;
}

bool color(position p) {
    assert(!is_empty(p));
    return p >> PIECE_BITS;
}

uint8_t piece(position p) {
    assert(!is_empty(p));
    return p & PIECE_MASK;
}

position board_position(
    const struct board *b,
    uint8_t x, uint8_t y
) {
    const position ret =
        b->v[(8 * y + x) >> 1];
    return (x & 1)
        ? position1(ret)
        : position0(ret);
}
```

Listing 8.5: Chess board storage

A bitmap can be used to store the presence or absence of a piece at a given position. The 64 positions fit an 8-byte (64-bit) integer. The same 4-bit piece data as before can follow the bitmap in order for each non-empty position, yielding a total of $64 + 4 \times 32 = 192$ bits $= 24$ bytes for the initial configuration of the board. Listing 8.6 shows this implementation. All omitted declarations and definitions remain the same, except for the enumeration, which now no longer has an `EMPTY` element.

```
struct board { uint64_t bitset; position v[32]; };

bool is_empty(const struct board *b, uint8_t x, uint8_t y) {
    const unsigned i = 8u * y + x;
    assert(i < 64);
    return !((b->bitset >> i) & 1);
}

position board_position(const struct board *b, uint8_t x, uint8_t y) {
    assert(!is_empty(b, x, y));
    const unsigned bi = 8u * y + x;
    const uint64_t mask = ((uint64_t)1 << bi) - 1;
    const unsigned i = popcnt(b->bitset & mask);
    const position ret = b->v[i >> 1];
    return (i & 1) ? position1(ret) : position0(ret);
}
```

Listing 8.6: Chess board storage with bit set

2.1.

The expression `m = b + (e - b) / 2` is a bit unusual but a mathematically correct way to obtain a value that is exactly between `b` and `e`. In the context of a binary search, we are dealing with integer indices, not continuous variables, so it must be proven that the expression correctly partitions the half-closed interval in two for every possible value of `b` and `e`.

When the length of the range is even, the expression is no different than the equivalent mathematical expression. Partitioning the array `[0, 1, 2, 3]`, assuming for simplicity in all examples that `b` is the beginning of the array at address `0`, gives:

$$\begin{aligned} m &= b + \frac{e-b}{2} \\ &= 0 + \frac{4-0}{2} \\ &= \frac{4}{2} = 2 \end{aligned}$$

$$|m-b| = |e-m| = 2$$

While not possible due to the loop termination condition in this case, the result is also correct for a range of length zero.

$$\begin{aligned} m &= 0 + \frac{0-0}{2} \\ &= |m-b| = |e-m| = 0 \end{aligned}$$

In the case of a range with odd length, e.g. `[0, 1, 2]`, one of the two sub-ranges has to have an extra element. Due to rounding, that is the second one:

$$\begin{aligned} m &= 0 + \frac{3-0}{2} \\ &= \left\lfloor \frac{3}{2} \right\rfloor = 1 \end{aligned}$$

$$|m-b| = 1 \qquad |e-m| = 2$$

We have therefore shown the following properties of the middle point expression for ranges of different lengths:

- Ranges of length zero remain the same.
- Ranges of even length are split evenly.
- Ranges of odd length are split into two ranges of even and odd length, respectively.
- For all lengths, the two resulting ranges are a sub-range of the original.

2.2.

There are two problems with this alternative expression. As we have seen in section 4.4 (*Undefined behavior*), pointer arithmetic is only valid inside the bounds of an array, up to and including one past its end. Therefore the expression `b + e` is undefined since it is outside the defined bounds for any range where `b` is not zero (practically, no object is ever stored at address zero in most architectures).

Even if that were not the case, a second problem occurs when either `b` or `e` are close to the end of the range of memory addresses. Assuming an architecture where pointers and integers are interconvertible for illustrative purposes, if `e == (void*)SIZE_MAX`, any non-zero value of `b` will result in overflow, yielding an incorrect result after the division and addition.

This problem does not exist in the original expression, as `e - b <= e` and therefore a valid value as long as the value of the difference can be represented by the `ptrdiff_t` type.

2.3.

Binary search, lower bound, and upper bound are very similar algorithms, so writing one after having written one of the others is simple, but rewriting `binary_search` requires careful attention to the invariants.

All three have the same iteration pattern: recursive binary partition based on a predicate. The crucial difference is in the comparisons that determine the partitions. As hinted in the exercise description, both lower and upper bound allow us to perform a single comparison per iteration and to define the entire algorithm in terms of a single comparison function, which makes writing generic code (v. section 5.2, *Metaprogramming*) easier.

Let us consider the lower bound first. We note that the conditional where `*m` and `x` are tested for equality has to be removed, as the search must proceed in that case if we are to find the leftmost occurrence. It may be surprising is that this is *all* that is needed to implement the algorithm, but if follows naturally from the adjusted loop invariants.

```
int *lower_bound(int x, int *b, int *e) {
    assert(is_sorted(b, e));
    int *const ib = b, *const ie = e;
    while(b != e) {
        assert(ib <= b && b <= e && e <= ie);
        assert(!lsearch(x, ib, b));
        assert(b == ib || b[-1] < x);
        assert(e == ie || !(*e < x));
        int *const m = b + (size_t)(e - b) / 2;
        if(*m < x)
            b = m + 1;
        else
            e = m;
    }
```

```
    assert(ib <= b && b == e && b <= ie);
    assert(b == ib || b[-1] < x);
    assert(b == ie || !(*b < x) || !lsearch(x, ib, ie));
    return b;
}
```

A few notable changes:

- As mentioned previously, there is now a single comparison per iteration.
- Furthermore, all comparisons, including in the invariants, use a single operation: less than (this is the reason for the unusual spellings such as `!(*e < x)`.
- The invariant involving `e` now allows for `*e == x`. Similarly, the invariant which stated that `!lsearch(x, e, ie)` had to be removed. In both cases, this is because duplicate elements are now allowed in the upper region.

From the last two loop invariants we know that when `b` is moved, it will never point to a value that is greater than `x`. Similarly, when `e` is moved, it will never point to a value that is less than `x`. Then, the intersection of the two, where `b` is moved to `e` in the final iteration of the loop (or when `e` is moved to `b`, if `x` is smaller than all values in the range), must be the lower bound.

Computing the upper bound follows a similar process. All that is required is to also choose the upper region of the range when `*m == x`. We spell the condition as `!(x < *m)` instead of `*m <= x` for the same reason cited above.

```
#include <assert.h>
#include <stdbool.h>
#include <stddef.h>

bool is_sorted(const int *b, const int *e);
bool lsearch(int x, const int *b, const int *e);

int *upper_bound(int x, int *b, int *e) {
    assert(is_sorted(b, e));
    int *const ib = b, *const ie = e;
    while(b != e) {
        assert(ib <= b && b <= e && e <= ie);
        assert(!lsearch(x, e, ie));
        assert(b == ib || !(x < b[-1]));
        assert(e == ie || x < *e);
        int *const m = b + (size_t)(e - b) / 2;
        if(!(x < *m))
            b = m + 1;
        else
            e = m;
    }
    assert(ib <= b && b == e && b <= ie);
    assert(b == ib || !(x < b[-1]));
    assert(b == ie || x < *e || !lsearch(x, ib, ie));
    return b;
}
```

To rewrite `binary_search`, we note that the return value of `lower_bound` must be one of three cases:

- `x` is larger than all values in the range. The lower bound is `e`.
- `x` is in the range. The element returned is equal to it.

- x is not in the range but smaller than some of the values in it. The element returned is the beginning of the sequence that is greater than it.

```
bool bsearch(int x, int *b, int *e) {
    b = lower_bound(x, b, e);
    return b != e && !(x < *b);
}
```

Similarly for upper_bound:

```
bool bsearch(int x, int *b, int *e) {
    e = upper_bound(x, b, e);
    return e != b && !(e[-1] < x);
}
```

3.1

There are several ways to conceptualize the operations involved in the reversal of a list. One of them is to start with an empty list and continuously move one element from the front of the old list to the front of the new one. Always pushing to the front of the new list effectively reverses the order of nodes.

```
struct node *list_reverse(struct node *n) {
    struct node *ret = NULL;
    for(struct node *next = NULL; n; n = next) {
        next = n->next;
        list_push_front(&ret, n);
    }
    return ret;
}
```

3.2

This exercise shows the benefit of treating the list head and links homogeneously. The function is passed a double pointer, so it can examine the value in the node via a double dereference, but also modify the head or node link pointer through it. Figures 8.1 and 8.2 show both cases.

```
struct node *list_remove(struct node **l, int value) {
    for(; *l; l = &(*l)->next) {
        if((*l)->value != value)
            continue;
        struct node *ret = *l;
        *l = ret->next;
        return ret;
    }
    return NULL;
}
```

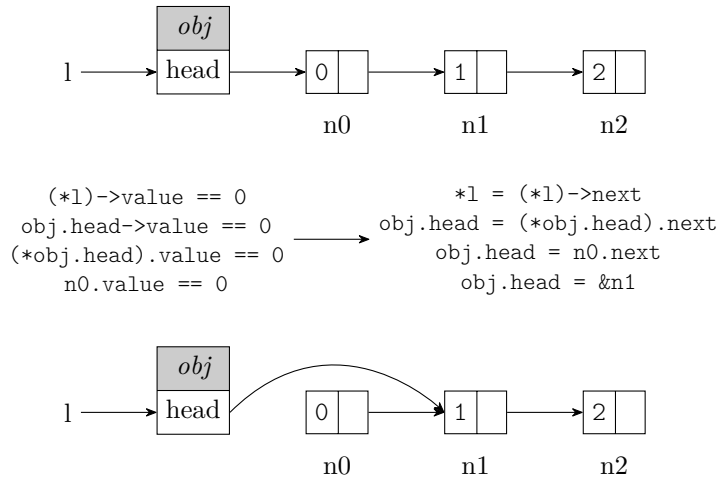


Figure 8.1: `list_remove(&l, 0)`

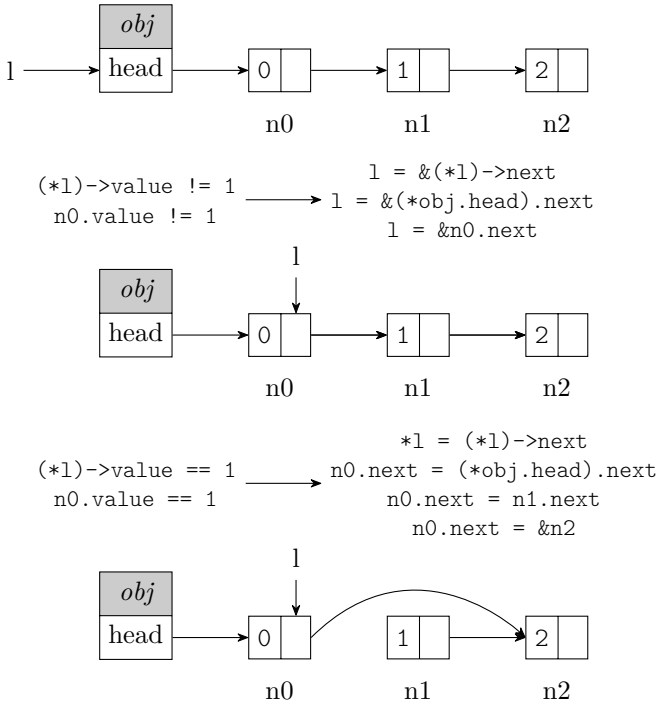


Figure 8.2: `list_remove(&l, 1)`

Listings

Bibliography

Brown, Neil (2009). “Linux kernel design patterns”. In: https://lwn.net/Articles/336224.

— (2011). “Object-oriented design patterns in the kernel”. In: https://lwn.net/Articles/444910.

Carruth, Chandler (2016). *Garbage In, Garbage Out: Arguing about Undefined Behavior*. https://www.youtube.com/watch?v=yG1OZ69H_-o.

Dietz, Will et al. (2012). “Understanding Integer Overflow in C/C++”. In: *Proceedings of the 34th International Conference on Software Engineering*. https://www.cs.utah.edu/~regehr/papers/overflow12.pdf.

Dijkstra, Edsger W. (1982). “Why numbering should start at zero”. In: https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF.

Drepper, Ulrich (2011). “Futexes Are Tricky”. In: https://akkadia.org/drepper/futex.pdf.

Edge, Jake (2015). “Inheriting capabilities”. In: https://lwn.net/Articles/632520.

ISO/IEC 9899:2011 (2011). C11 Standard, https://www.iso.org/standard/57853.html.

Itanium C++ ABI (2017). https://itanium-cxx-abi.github.io/cxx-abi.

Kerrisk, Michael (2010). *The Linux Programming Interface*. ISBN-13: 978-1-59327-220-3, https://man7.org/tlpi/. No Starch Press.

Lamport, Leslie (1978). “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM*. http://lamport.azurewebsites.net/pubs/time-clocks.pdf.

— (1979). “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers*. http://lamport.azurewebsites.net/pubs/multi.pdf.

Lions, John (1977). *A Commentary on the Sixth Edition Unix Operating System*. ISBN-13: 978-1-57398-013-5. University of New South Wales.

Matz, Michael et al. (2012). *System V Application Binary Interface. AMD64 Architecture Processor Supplement*. https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf.

Preshing, Jeff (2011). *Locks Aren't Slow; Lock Contention Is*. https://preshing.com/20111118/locks-arent-slow-lock-contention-is/.

Ritchie, Denis M. (1996). “The Development of the C Language”. In: *History of Programming Languages*. https://www.bell-labs.com/usr/dmr/www/chist.html.

Stroustrup, Bjarne (1989). “Multiple Inheritance for C++”. In: *Computing Systems*. https://www.usenix.org/legacy/publications/compsystems/1989/fall_stroustrup.pdf.

Summit, Steve (1995). *comp.lang.c Frequently Asked Questions*. http://c-faq.com.

The as-if rule (n.d.). https://en.cppreference.com/w/cpp/language/as_if.

The Open Group (1997). “64-Bit Programming Models: Why LP64?” In: https://unix.org/version2/whatsnew/lp64_wp.html.