Введение в C++

Языки программирования C и C++

- **Не существует языка программирования C/C++. Есть два разных языка.**
  - Они разрабатываются разными группами людей.
  - Они имеют разные компиляторы.
  - Хотя эти языки имеют общие базовые конструкции, стиль и приёмы программирования на них сильно отличаются.
  - Один язык не является подмножеством другого. Случайно взятая программа на C вряд ли окажется валидной программой на C++.
    - Правда, модификации требуемые чтобы сделать её валидной программой на C++ небольшие.

Пример отличий между C и C++

```
a ? b : c = 42


a ? b : (c = 42)          // C++
(a ? b : c) = 42          // C
```

Целочисленные типы

- Целочисленные типы записываются несколькими ключевыми словами
  - `signed`/`unsigned` обозначают знаковость
  - `short`/`long` для обозначают размер
  - `int`

Целочисленные типы

	знаковый	беззнаковый
`short`	`short`	`unsigned short`
(обычного размера)	`int`	`unsigned`
`long`	`long`	`unsigned long`
`long long`	`long long`	`unsigned long long`

	знаковый		беззнаковый
символьный	`signed char`	`char`	`unsigned char`

Эквивалентность типов

Как проверить что два типа одинаковые? Использовать перегрузку.

```
void f(int a)
{}

void f(signed int a)
{}
```

```
4:6: error: redefinition of 'void f(int)'
    4 | void f(signed int a)
      |      ^
1:6: note: 'void f(int)' previously defined here
    1 | void f(int a)
      |      ^
```

```
void f(char a)
{}

void f(signed char a)
{}
```

Целочисленные типы

Тип	32-бит	64-бит Linux	64-бит Windows
`char`	1 байт	1 байт	1 байт
`short`	2 байта	2 байта	2 байта
`int`	4 байта	4 байта	4 байта
`long`	4 байта	8 байтов	4 байта
`long long`	8 байтов	8 байтов	8 байтов

#include <cstdint>

- **Предоставляет типы:**
  - `int8_t`, `uint8_t`
  - `int16_t`, `uint16_t`
  - `int32_t`, `uint32_t`
  - `int64_t`, `uint64_t`

#include <cstdint>

```
typedef char           int8_t;
typedef unsigned char  uint8_t;
typedef short          int16_t;
typedef unsigned short uint16_t;
typedef int            int32_t;
typedef unsigned int   uint32_t;
typedef long           int64_t;
typedef unsigned long  uint64_t;
```

Ещё один тип

Какой тип использовать для индексации в массиве?

```
for (??? i = 0; i != 100; ++i)
    arr[i] = 42;
```

Ещё один тип

Какой тип использовать для индексации в массиве?

```
for (size_t i = 0; i != 100; ++i)
    arr[i] = 42;
```

	signed	unsigned
Индекс в массиве	`ptrdiff_t`	`size_t`

Алгоритм выбора типа

- Правда ли, что значение приходит из какой существующей функции?
  - Если да, пишем тот же тип, что и в сигнатуре функции. Не надо делать лишние конверсии, поскольку каждая конверсия это потенциально потерянные данные.
- Правда ли, что это индекс в массиве либо число объектов в памяти?
  - Если да, пишем `size_t` или `ptrdiff_t`.
- Иначе выбираем тип `[u]intN_t` в соответствии с предметной областью.

Алгоритм выбора типа

Q: Что если мне нужно просто `int`?

A: Просто `int` в компьютерах не бывает. Они все имеют определенную битность.

Алгоритм выбора типа

Приведённый алгоритм неполный. Какой тип выбрать здесь?

```
? max(? a, ? b)
{
    return a < b ? b : a;
}
```

Алгоритм выбора типа

Если функция применима к любому типу возможно её можно сделать шаблонной:

```
template< class T >
T max(T a, T b)
{
    return a < b ? b : a;
}
```

Алгоритм выбора типа

Если необходимо перечислить все типы следует использовать встроенные типы, а не типы из `<cstdint>`:

`char`	`unsigned char`	`signed char`	`int8_t`	`uint8_t`
`short`	`unsigned short`		`int16_t`	`uint16_t`
`int`	`unsigned`		`int32_t`	`uint32_t`
`long`	`unsigned long`		`int64_t`	`uint64_t`
`long long`	`unsigned long long`			

Алгоритм выбора типа

Перечисление типов может быть необходимо при написание traits-классов. Например так написан **`std::numeric_limits<T>`**:

```
int64_t m = std::numeric_limits<int64_t>::max();
```

Числа с плавающей точкой

При вычислениях люди используют экспоненциальную запись вида $m \cdot 10^n$. Например:

- $6.02214076 \cdot 10^{23}$
- $1.60217662 \cdot 10^{-19}$

В компьютерах используется тоже самое только в двоичной системе счисления:

- $1.0000 \cdot 2^0 = 1$
- $1.1000 \cdot 2^0 = 1.5$
- $1.1010 \cdot 2^0 = 1.625$
- $1.1011 \cdot 2^0 = 1.6875$

Числа с плавающей точкой

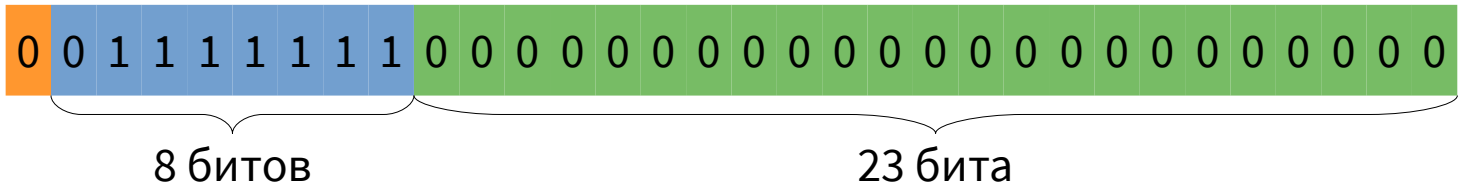
Обратим внимание, что в двоичной системе счисления первая цифра всегда 1:

- $1.0000 \cdot 2^0 = 1$
- $1.1000 \cdot 2^0 = 1.5$
- $1.1010 \cdot 2^0 = 1.625$
- $1.1011 \cdot 2^0 = 1.6875$

Кроме нуля. Ноль имеет особое представление.

Числа с плавающей точкой

$+1.0000 \cdot 2^{0}$



Составные части называются мантисса, экспонента, знак.

Числа с плавающей точкой

Точность	GCC	MSVC	Размер мантиссы	Размер экспоненты
Половинная (Half)	__fp16	N/A	11	5
bfloat16	N/A	N/A	7	8
Одинарная (Single)	float	float	23	8
Двойная (Double)	double	double long double	52	11
Расширенная (Extended)	long double	N/A	63+1	15
Четверная (Quadruple)	__float128	N/A	112	15

Примеры кодирования

1

Примеры кодирования

1

2 ?

Примеры кодирования

1 

2 

3 ?

Примеры кодирования

1 

2 

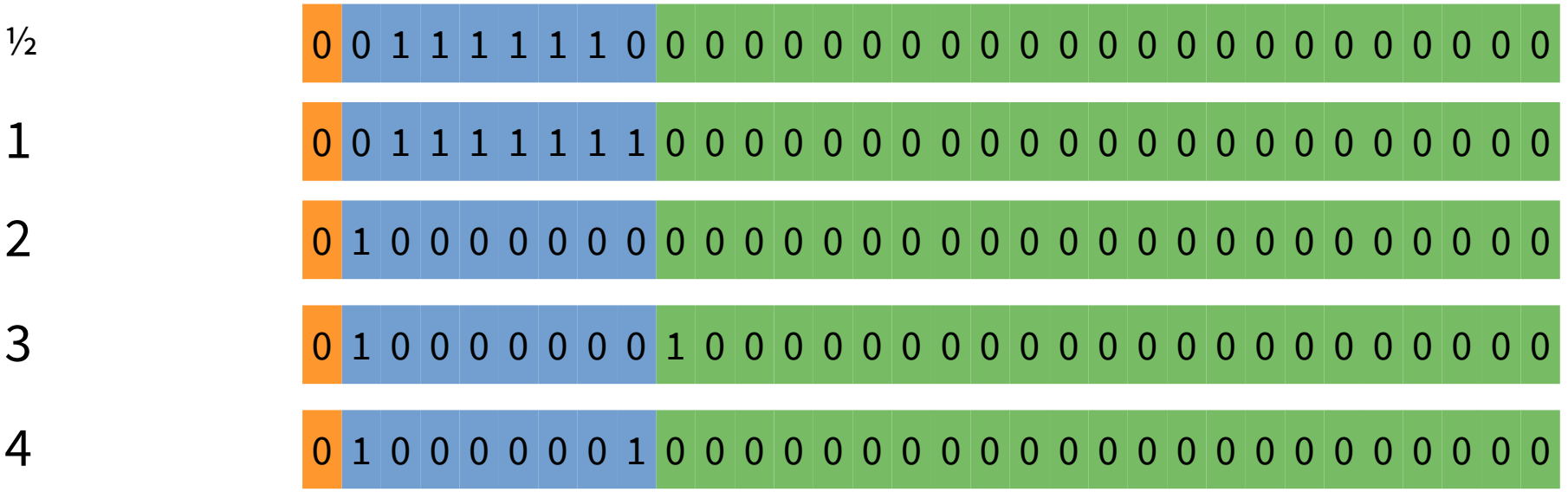
3 

4 ?

Примеры кодирования

½	?
1	0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3	0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4	0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

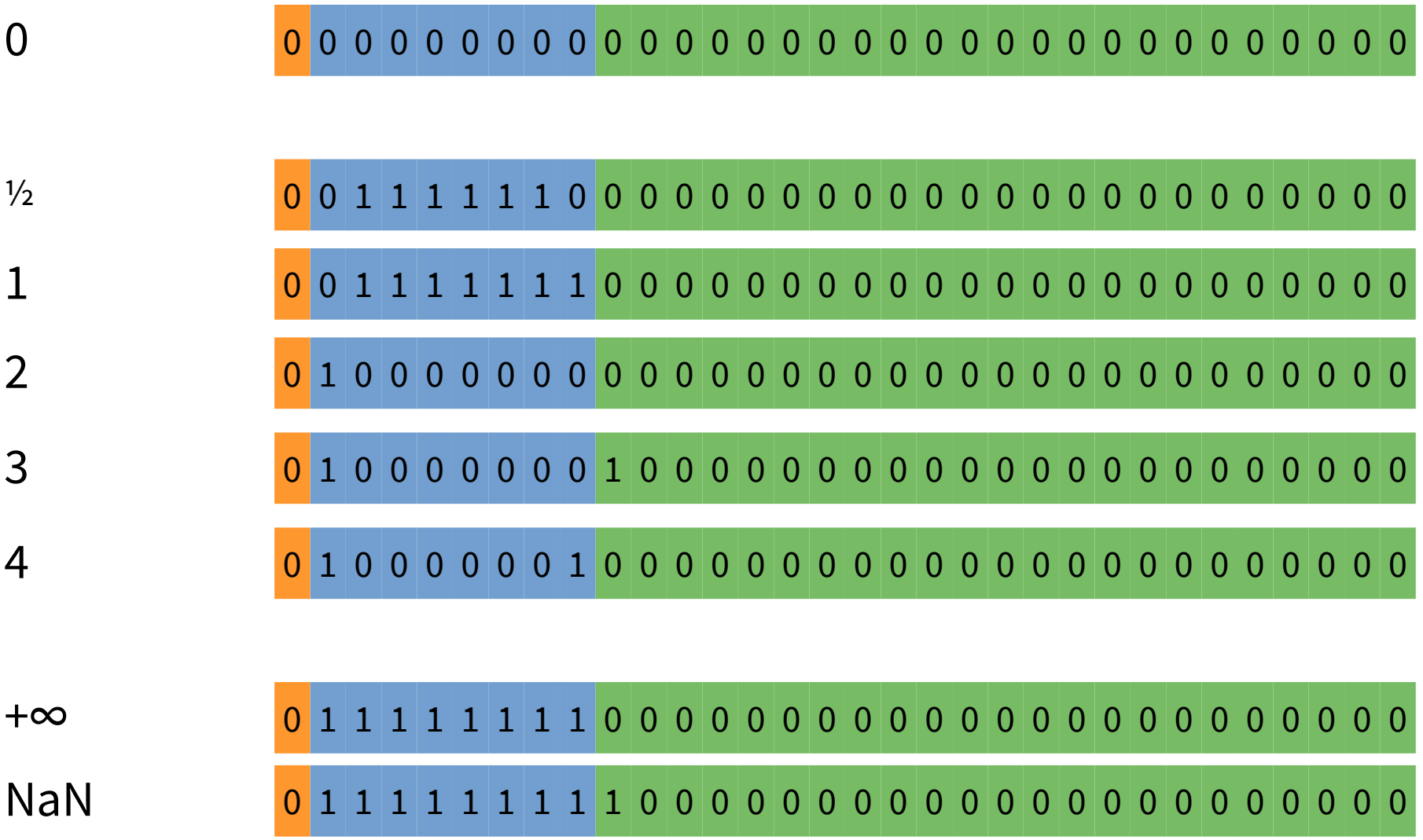
Примеры кодирования



Примеры кодирования

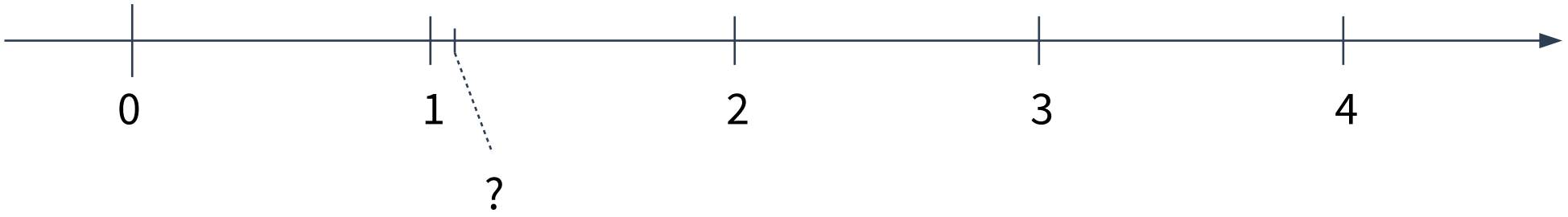
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
½	0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1	0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3	0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4	0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Примеры кодирования

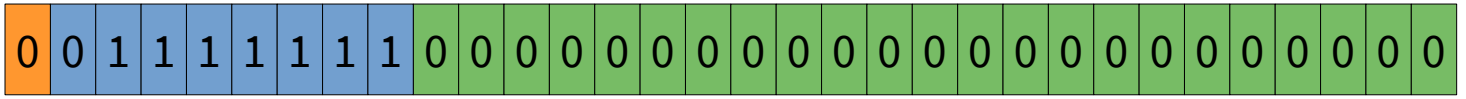


Расположение чисел с плавающей точкой

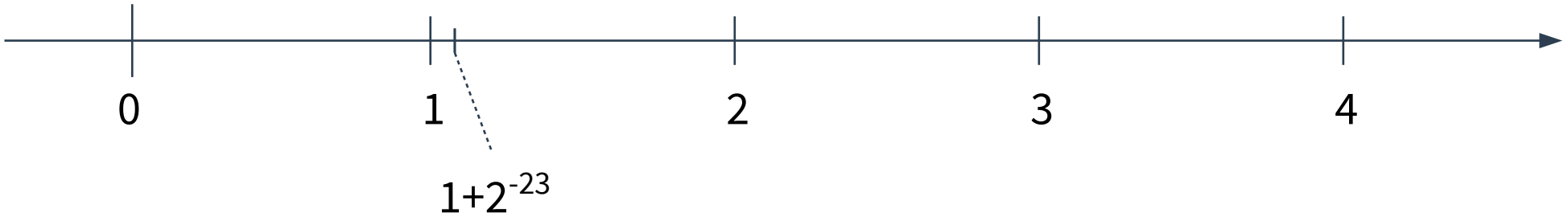
Расположение чисел с плавающей точкой

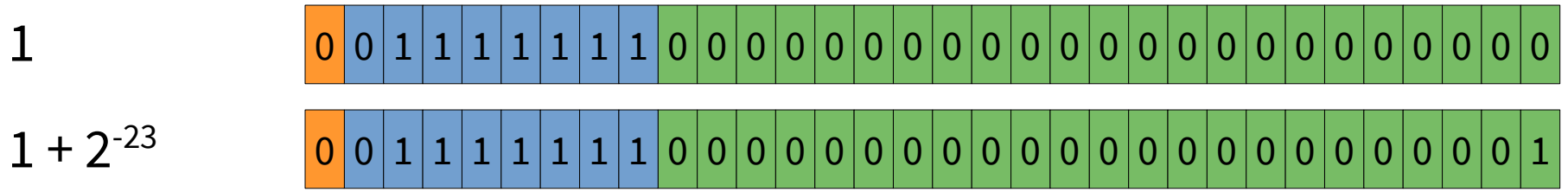


1

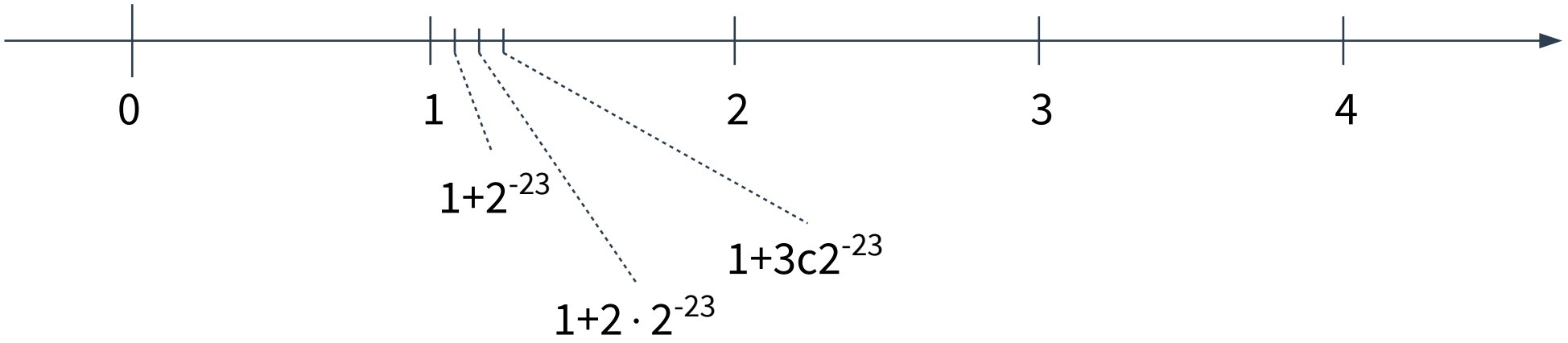


Расположение чисел с плавающей точкой

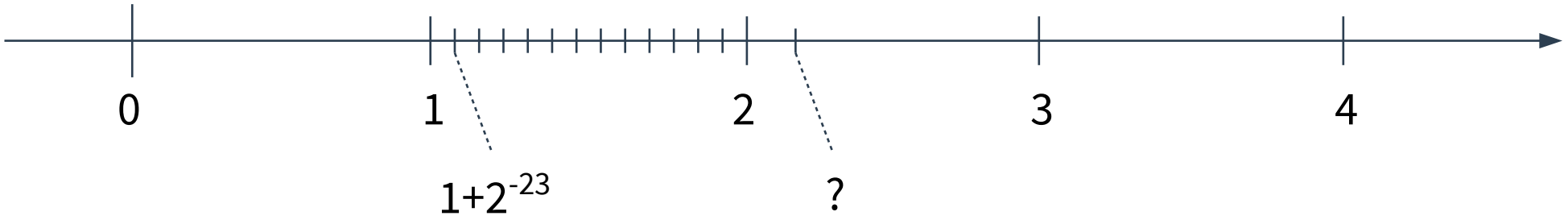




Расположение чисел с плавающей точкой



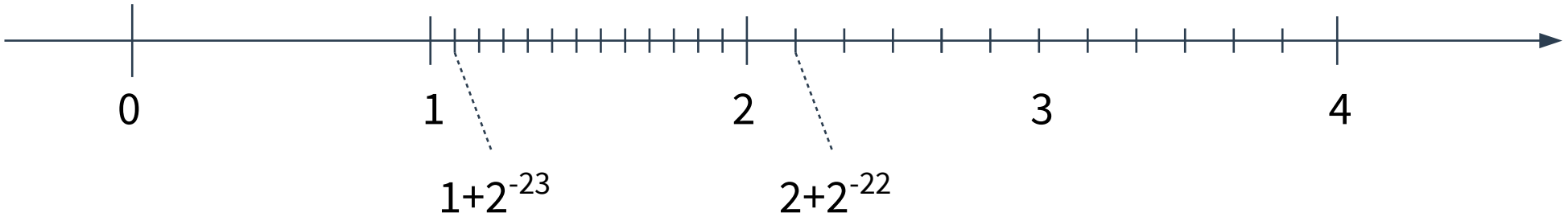
Расположение чисел с плавающей точкой



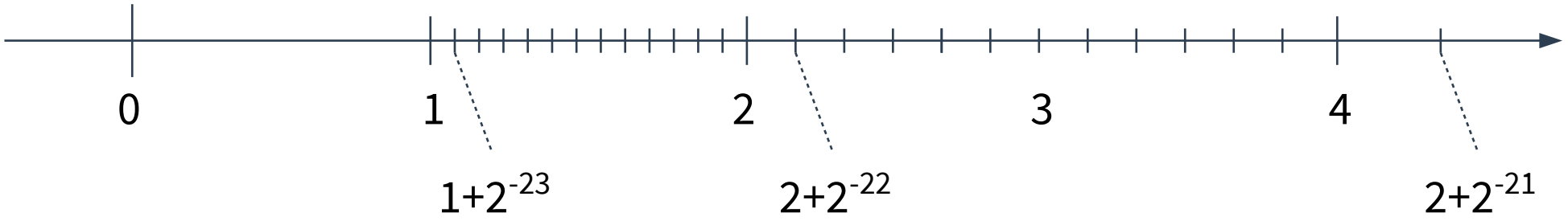
2 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

? 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

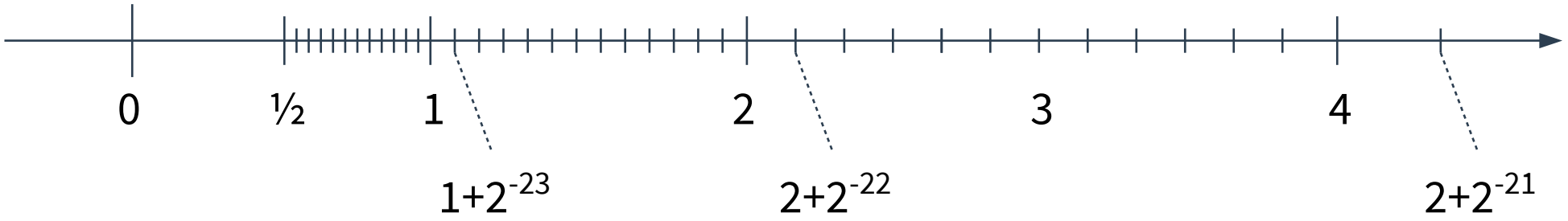
Расположение чисел с плавающей точкой



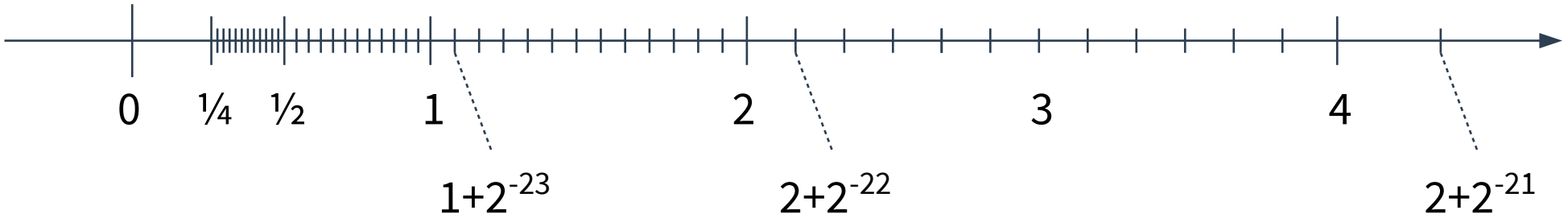
Расположение чисел с плавающей точкой



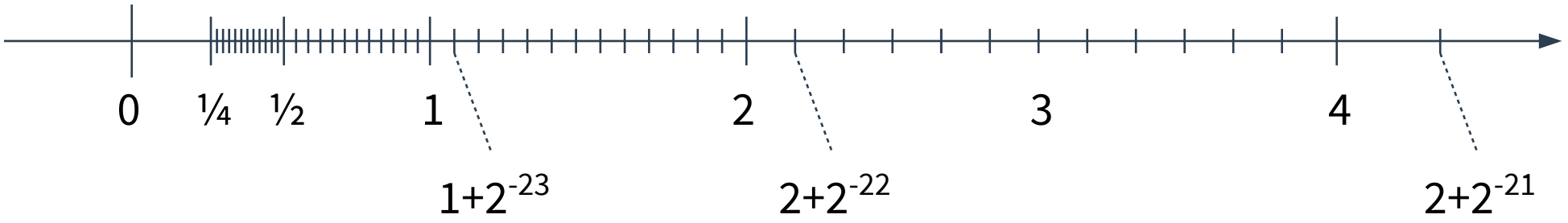
Расположение чисел с плавающей точкой



Расположение чисел с плавающей точкой



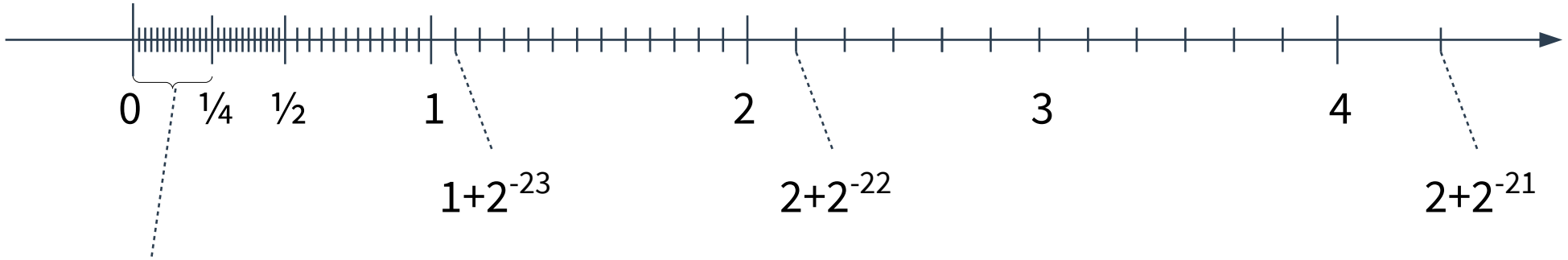
Денормализованные числа




```
if (a != b)
    c / (a – b)
```

Можно ли добавить чисел между нулём и минимальным, чтобы избежать деления на 0?

Денормализованные числа



Денормализованные числа

denorm_min	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
min	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Операции

- Каждое число с плавающей точкой соответствует конкретному вещественному числу.
- IEEE 754 определяет операции +, -, $\cdot$, /, $\sqrt{}$.
  - Они вычисляют результат в вещественных числах, а затем округляют его к ближайшему представимому числу с плавающей точкой в соответствии с текущим режимом округления.
- Режимы округления:
  - К ближайшему представимому (используется по умолчанию)
  - К нулю
  - К минус бесконечности
  - К плюс бесконечности

Любопытный пример

```
#include <iostream>


int main()
{
    bool x = (0.1 * 3) == 0.3;
    std::cout << std::boolalpha << x;
}
```

Любопытный пример

```
#include <iostream>

int main()
{
    bool x = (0.1 * 3) == 0.3;
    std::cout << std::boolalpha << x;
}

$ ./a.out
false
```

Любопытный пример

```
#include <iostream>
#include <iomanip>

int main()
{
    std::cout << std::setprecision(1000);

    std::cout << "0.1     = " << 0.1 << '\n';
    std::cout << "0.1 * 3 = " << 0.1 * 3 << '\n';
    std::cout << "0.3     = " << 0.3 << '\n';
}

$ ./a.out
0.1     = 0.1000000000000000055511151231257827021181583404541015625
0.1 * 3 = 0.3000000000000000444089209850062616169452667236328125
0.3     = 0.2999999999999999888977697537484345957636831919091796875
```

Любопытный пример

```
$ ./a.out
0.1     = 0.1000000000000000055511151231257827021181583404541015625
0.1 * 3 = 0.3000000000000000444089209850062616169452667236328125
0.3     = 0.2999999999999999888977697537484345957636831991796875
```

0.1

Shortest round-trip conversion

Свойства

- Выполняется ли для чисел с плавающе точкой коммутативность? $a + b = b + a$

Свойства

- Выполняется ли для чисел с плавающе точкой коммутативность? `a + b = b + a`
- А ассоциативность? `(a + b) + c = a + (b + c)`

Сравнения

Потеря точности

```
double f(double x)
{
    return x * x + sqrt(x);
}


double f_inverse(double c)
{
    double u = cbrt(sqrt(1.0 / 27.0 * pow(c, 3) + 1.0 / 256.0) + 1.0 /
16.0);
    double v = -2.0 / 3.0 * c / u + 2 * u;
    double w = -0.5 * sqrt(v) + 0.5 * sqrt(-v + 2 / sqrt(v));
    return w * w;
}
```

Потеря точности

```
double f_inverse(double c)
{
    double u = cbrt(sqrt(1.0 / 27.0 * pow(c, 3) + 1.0 / 256.0) + 1.0 / 16.0);
    double v = -2.0 / 3.0 * c / u + 2 * u;
    double w = -0.5 * sqrt(v) + 0.5 * sqrt(-v + 2 / sqrt(v));
    return w * w;
}
```

x = 80000

c = 6400000282.8427124

u = 46188.022555790791

v = 7.2759576141834259e-11

w = 242.10984763985175

Потеря точности

```
double f_inverse(double c)
{
    double u = cbrt(sqrt(1.0 / 27.0 * pow(c, 3) + 1.0 / 256.0) + 1.0 / 16.0);
    double v = -2.0 / 3.0 * c / u + 2 * u;
    double w = -0.5 * sqrt(v) + 0.5 * sqrt(-v + 2 / sqrt(v));
    return w * w;
}
```

x = 80000

c = 6400000282.8427124

u = 46188.022555790791

v = 7.2759576141834259e-11

w = 242.10984763985175

Потеря точности

```
double f_inverse(double c)
{
    double u = cbrt(sqrt(1.0 / 27.0 * pow(c, 3) + 1.0 / 256.0) + 1.0 / 16.0);
    double v = -2.0 / 3.0 * c / u + 2 * u;
    double w ≈ sqrt(2 / sqrt(v));
    return w * w;
}
```

x = 80000

c = 6400000282.8427124

u = 46188.022555790791

v = 7.2759576141834259e-11

w = 242.10984763985175

Потеря точности

```
double f_inverse(double c)
{
    double u ≈ sqrt(c/3);
    double v = -2.0 / 3.0 * c / u + 2 * u;
    double w ≈ sqrt(2 / sqrt(v));
    return w * w;
}
```

x = 80000

c = 6400000282.8427124

u = 46188.022555790791

v = 7.2759576141834259e-11

w = 242.10984763985175

Потеря точности

```
double f_inverse(double c)
{
    double u ≈ sqrt(c/3);
    double v ≈ -2.0 / 3.0 * c * sqrt(3/c) + 2 * sqrt(c/3)
             ≈ -2 * sqrt(c/3) + 2 * sqrt(c/3)
             ≈ 0;
    double w ≈ sqrt(2 / sqrt(v));
    return w * w;
}
```

```
x = 80000
c = 6400000282.8427124
u = 46188.022555790791
v = 7.2759576141834259e-11
w = 242.10984763985175
```

Enum (1)

```
enum class color
{
    red,
    green,
    blue
};

void draw(color);

int main()
{
    draw(color::green);
}
```

Enum (2)

Зачем нужны enum'ы? Почему не пользоваться просто целочисленными константами?

```
enum class color
{
    red,
    green,
    blue
};

void draw(color);

int main()
{
    draw(color::green);
}
```

```
int32_t const RED = 0;
int32_t const GREEN = 1;
int32_t const BLUE = 2;

void draw(int32_t);

int main()
{
    draw(GREEN);
}
```

Enum (3)

- Типизация
  - `void draw(color, pen_style, brush_style);`
  - https://radekvit.medium.com/writing-interfaces-stay-true-in-booleans-ed52b5f1b720

Enum (4)

- Типизация
  - `void draw(color, pen_style, brush_style);`
  - https://radekvit.medium.com/writing-interfaces-stay-true-in-booleans-ed52b5f1b720
- Логическая группировка

Enum (5)

```
int32_t const no_pen = 0;
int32_t const solid_line = 1;
int32_t const dash_line = 2;
int32_t const dot_line = 3;
int32_t const dash_dot_line = 4;
int32_t const dash_dot_dot_line = 5;
int32_t const custom_dash_line = 6;

int32_t const no_brush = 0;
int32_t const solid_pattern = 1;
int32_t const dense_pattern = 2;
int32_t const cross_pattern = 3;
int32_t const linear_gradient_pattern = 4;
```

Enum (6)

```
int32_t const no_pen = 0;
int32_t const solid_line = 1;
int32_t const dash_line = 2;
int32_t const dot_line = 3;
int32_t const dash_dot_line = 4;
int32_t const dash_dot_dot_line = 5;
int32_t const custom_dash_line = 6;

int32_t const no_brush = 0;
int32_t const solid_pattern = 1;
int32_t const dense_pattern = 2;
int32_t const cross_pattern = 3;
int32_t const linear_gradient_pattern = 4;
```

```
enum class pen_style
{
    no_pen,
    solid_line,
    dash_line,
    dot_line,
    dash_dot_line,
    dash_dot_dot_line,
    custom_dash_line,
};

enum class brush_style
{
    no_brush,
    solid_pattern,
    dense_pattern,
    cross_pattern,
    linear_gradient_pattern,
};
```

Enum (7)

enum’ы компилируются и работают просто как числа. Называется underlying type. По-умолчанию это int. Но можно указать другой тип.

```
enum class color : uint8_t
{
    // ...
};
```

Enum (8)

По умолчанию enumerator’ы имеют значение, на 1 больше предыдущего, но возможно задать значение явно:

```
enum class color : uint8_t
{
    blue  = 1,
    green = 2,
    red   = 4,
};

enum -> underlying type
    static_cast<int8_t>(color::green);
    std::to_underlying(c);

underlying type -> enum
    static_cast<color>(2);
```

Enum (9)

```
enum class color
{
    red,
    green,
    blue
};

int main()
{
    color a = color::green;
    int b = static_cast<int>(a);
}
```

```
enum color
{
    red,
    green,
    blue
};

int main()
{
    color a = green;
    int b = a;
}
```

Классы (1)

Классы используются чтобы группировать данные:

```
struct point
{
    float x;
    float y;
    float z;
};

int main()
{
    point p = {1, 0, 0};
    float t = p.x;
}
```

Классы (2)

- Пример вверху взят из реальной программы.
- Имея подходящую абстракцию его можно заменить на строчку внизу.

```
p[0] = 0;
p[1] = fabs( r );
p[2] = -dist;

projected[0] = p[0] * tr.viewParms.projectionMatrix[0] +
               p[1] * tr.viewParms.projectionMatrix[4] +
               p[2] * tr.viewParms.projectionMatrix[8] +
               tr.viewParms.projectionMatrix[12];

projected[1] = p[0] * tr.viewParms.projectionMatrix[1] +
               p[1] * tr.viewParms.projectionMatrix[5] +
               p[2] * tr.viewParms.projectionMatrix[9] +
               tr.viewParms.projectionMatrix[13];

projected[2] = p[0] * tr.viewParms.projectionMatrix[2] +
               p[1] * tr.viewParms.projectionMatrix[6] +
               p[2] * tr.viewParms.projectionMatrix[10] +
               tr.viewParms.projectionMatrix[14];

projected[3] = p[0] * tr.viewParms.projectionMatrix[3] +
               p[1] * tr.viewParms.projectionMatrix[7] +
               p[2] * tr.viewParms.projectionMatrix[11] +
               tr.viewParms.projectionMatrix[15];
```

```
float4 projected = tr.viewParms.projectionMatrix
                 * float4(0.f, fabs(r), -dist, 1.f);
```

Классы (3)

- Группировка данных
  - Уменьшает boilerplate
  - Снижает вероятность ошибки (x, y, z) → (x, y, y)

Классы (4)

Классы можно использовать, чтобы возвращать несколько значений из функции.

```
#include <algorithm>
#include <cstdint>
#include <vector>

void test()
{
    std::vector<int32_t> v = { 4, 8, 15, 16, 23, 42 };

    std::ranges::minmax_result r = std::ranges::minmax(v);
    r.min;
    r.max;
}
```

Классы (5)

Классы можно использовать, если нужно передать много параметров в функцию.

https://www.reddit.com/r/cpp/comments/m2tgx1/comment/gqnw0jf/

```
struct tea_input {
    TeaType type;
    bool milk;
    bool lemon;
    bool sugar;
    bool leave_bag;
};

Beverage MakeTea(const tea_input&);

auto beverage = MakeTea({
    .type = TeaType::EarlGrey,
    .milk = true,
    .lemon = false,
    .sugar = false,
    .leave_bag = true,
});
```

Const

```
double const PI = 3;
size_t const BUFFER_SIZE = 8192;



int32_t const NUMBERS[6] = {4, 8, 15, 16, 23, 42};



struct complex
{
    double re, im;
};

complex const C = {0.866, 0.5};
```

Const (2)

```
size_t const BUFFER_SIZE = 8192;

int main()
{
    BUFFER_SIZE = 8193; // ERROR
}
```

```
error: assignment of read-only variable
‘BUFFER_SIZE’
    7 |     BUFFER_SIZE = 8193;
      |     ~~~~~~~~~~~~^~~~~~
```

Const (3)

```
double const PI = 3;
```

What is the type of **`&PI`**?

Const (3)

```
double const PI = 3;
```

What is the type of **`&PI`**?

- **`double*`** causes problems:

```
double* ptr = &PI;
*ptr = 4;
```

Const (4)

```
double const PI = 3;

double const* ptr = &PI;
*ptr = 4; // error
```

Const (5)

A pointer is a object too. Const is applicable to them too:

```
int32_t a = 42;
int32_t b = 42;

int32_t* p = &a;
p = &b; // OK

int32_t* const q = &a;
q = &b; // ERROR
```

Const (6)

```
int32_t       *       a = nullptr;
int32_t       * const b = nullptr;
int32_t const*       c = nullptr;
int32_t const* const d = nullptr;
```

`const` means pointer itself can not be modified.

- `ptr = ...` is forbidden.

`const` means the object the pointer points to can not be modified.

- `*ptr = ...` is forbidden.