

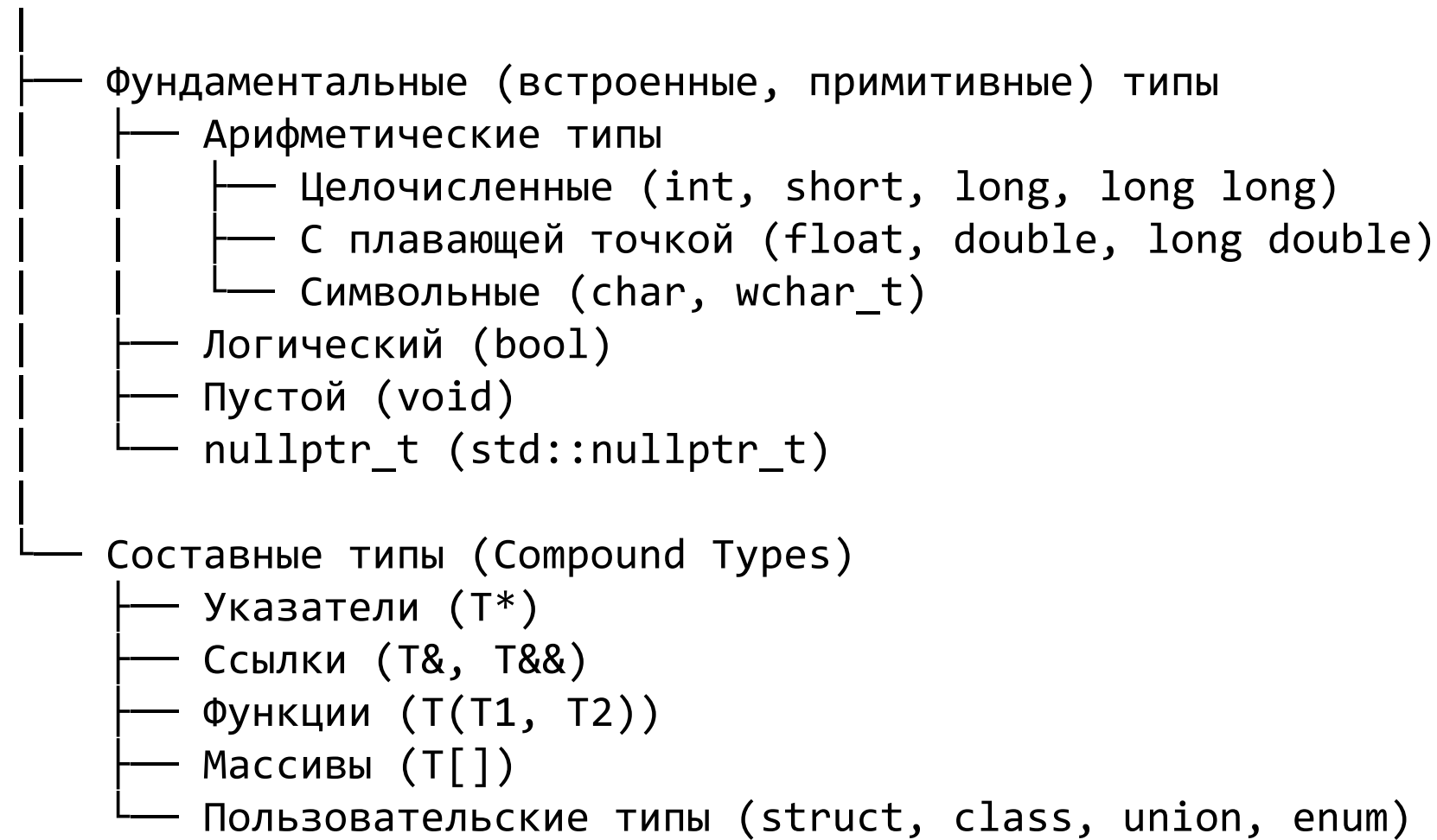
Программирование на языке C++

Лекция 6

C++

Типы данных C++

Типы данных C++



Типы данных C++

Тип определяет:

- Объем памяти, выделенный для переменной (или результата выражения);
- Диапазон значений, которые могут быть сохранены, и то, как компилятор представляет эти значения в побитовом виде;
- Операции, которые разрешены для этих объектов, и обеспечивает семантику;
- И др.

Фундаментальные типы. Типы полностью определяющие себя. Предоставляются самим языком программирования и не требующие дополнительных заголовков (библиотек).

Составные типы. Типы являются композицией других типов или ссылаются на другой тип.

Фундаментальные типы

Подробную справочную информацию по фундаментальным типам можно найти на: <https://en.cppreference.com/w/cpp/language/types.html>

Системно-зависимые псевдонимы

Системно-зависимые псевдонимы

- |— size_t, ptrdiff_t // Из <stddef>
- |— uintptr_t, intptr_t // Из <stdint>
- |— Фиксированные типы // Из <stdint>
- |— Типы максимальной длины // Из <stdint>

cstdint: <https://en.cppreference.com/w/cpp/types/integer.html>

stddef: <https://en.cppreference.com/w/cpp/header/cstddef.html>

Спецификаторы типа в C++

Спецификаторы типа в C++

Спецификаторы в C++

- |
- |— Квалификаторы типа (Type qualifiers) - часть системы типов
 - | |— const
 - | |— volatile
 - | |— mutable (только для членов класса/лямбда функций)
- |
- |— Спецификаторы вычислений (Evaluation specifiers) - C++11
 - | |— constexpr / consteval (C++20)
 - | |— constexpr (C++20)
- |
- |— Спецификаторы хранения (Storage class specifiers)
 - | |— auto (устаревшее)
 - | |— register (устаревшее)
 - | |— static
 - | |— extern
 - | |— thread_local

Квалификатор const

```
const /* имя типа*/ name = /* инициализатор */;  
/* имя типа*/ const name = /* инициализатор */;
```

Запрещает изменение значения переменной после создания;
Обязательно должен быть инициализатор:

```
const int i = 10;  
cout << i;      // Чтение - ОК  
i = a;          // Запись - Ошибка
```

Инициализатор может быть не известен на этапе компиляции:

```
int a;  
cin >> a;  
const int b = a;  // ОК
```


Квалификатор volatile

```
volatile /* имя типа*/ name;  
/* имя типа*/ volatile name;
```

Запрещает компилятору оптимизировать чтение/запись в переменную;
Можно комбинировать с `const`;

Квалификатор volatile

```
// БЕЗ volatile - компилятор может оптимизировать
int status = 0;

// Компилятор видит, что status не меняется в цикле
// Может преобразовать в бесконечный цикл!
while (status == 0) {
    // Здесь status мы не меняем
}
```

Квалификатор volatile

// С volatile - компилятор не оптимизирует доступ

```
volatile int status = 0;
```

// Компилятор ГАРАНТИРОВАННО будет читать status из памяти
каждый раз

```
while (status == 0) {
```

```
    // Ожидание изменения status (например, от аппаратуры)
```

```
}
```

const volatile

// С volatile - компилятор не оптимизирует доступ

```
const volatile int status = 0;
```

// Компилятор ГАРАНТИРОВАННО будет читать status из памяти каждый раз

```
while (status == 0) {
```

```
    // Ожидание изменения status (например, от аппаратуры)
```

```
}
```

Квалификатор mutable

```
int i = 0;
```

```
auto foo = [i]() {  
    i = 10;    // Ошибка. i - read-only  
};
```

```
auto foo = [i]() mutable {  
    i = 10;    // ОК, но меняется всё равно копия i  
};
```

```
// Подробнее будет рассмотрен когда дойдём до ООП
```

Препреквизит: Run time vs. Compile time

Временем выполнения (Run time) – от момента запуска программы до момента завершения.

Время компиляции (Compile time) – от начала сборки программы почти до получения исполняемого файла. В это время запущен и работает компилятор (это его время исполнения).

Часть вычислений из Run time можно перенести в Compile time (т.е. переложить на компилятор) для экономии ресурсов компьютера.

```
cout << 1 + 2; // Вычисление выполнит компилятор и заменит на 3
```

```
int a;
```

```
cin >> a;
```

```
cout << 1 + a; // Вычисление будет выполнять наша программа каждый раз
```

Спецификатор constexpr

```
constexpr /* имя типа*/ name = /* инициализатор */;  
/* имя типа*/ constexpr name = /* инициализатор */;
```

Запрещает изменение значения переменной после создания, поэтому обязательно должен быть инициализатор:

```
constexpr int i = 10;  
cout << i;      // Чтение - ОК  
i = a;          // Запись - Ошибка
```

Указывает компилятору, что значение переменной **должно** быть вычислено на этапе компиляции, поэтому инициализатор должен быть известен/вычислим на этапе компиляции:

```
int a;  
cin >> a;  
constexpr int b = a; // Ошибка. a неизвестно на этапе компиляции
```

Инициализатор может быть литералом или выражением и содержать `constexpr/constexpr` функции.

Спецификатор constexpr

// constexpr переменные

```
constexpr int size = 100;
```

```
constexpr double pi = 3.141592653589793;
```

```
constexpr int array_size = size * 2;
```

// Компилятор вычисляет значение

// Вычисляется на этапе компиляции

```
int two = 2;
```

```
constexpr double _2pi = pi * two;
```

// Ошибка

Спецификатор constexpr с функциями

Спецификатор `constexpr` с функциями позволяет использовать эту функцию для вычислений как на этапе компиляции, так и на этапе исполнения программы. Чтобы функция могла быть `constexpr` есть ряд ограничений.

`// constexpr функция`

```
constexpr int factorial(int n) {  
    return (n <= 1) ? 1 : n * factorial(n - 1);  
}
```

```
constexpr int fact_5 = factorial(5); // Вычисляется на этапе компиляции: 120  
int fact_6 = factorial(6); // Вычисляется на этапе компиляции: 720  
int n;  
cin >> n;  
int fact_n = factorial(n); // Вычисляется на этапе исполнения
```

Спецификатор consteval

Спецификатор `constexpr` может быть использован только с функциями и указывает, что данная функция должна возвращать константу времени компиляции.

Чтобы функция могла быть `constexpr` есть ряд ограничений.

`// constexpr функция`

```
constexpr int factorial(int n) {  
    return (n <= 1) ? 1 : n * factorial(n - 1);  
}
```

```
constexpr int fact_5 = factorial(5); // Вычисляется на этапе компиляции: 120  
int fact_6 = factorial(6); // Вычисляется на этапе компиляции: 720  
int n;  
cin >> n;  
int fact_n = factorial(n); // Запрещено
```

Спецификатор `constinit`

```
constinit /* имя типа*/ name = /* инициализатор */;  
constinit static /* имя типа*/ name = /* инициализатор */;
```

Указывает компилятору, что значение переменной **должно** быть вычислено на этапе компиляции;

Применяется только к глобальным или `static` переменным.

Инициализатор может содержать `constexpr/constexpr` функции;

В процессе работы программы не запрещает изменение значения;

```
static int i = 10;  
cout << i;    // Чтение - ОК  
i = 5;        // Запись - ОК
```

Спецификатор constinit

```
constexpr int calc() { return 5 * 5; }  
constinit int global = calc(); // Инициализация в compile-time  
  
void foo() {  
    constinit static int stat = calc(); // Инициализация в compile-time  
    constinit int local = calc();      // Ошибка  
  
    global = 42; // OK: переменная не константа!  
}
```

Спецификаторы хранения

Спецификаторы хранения

Спецификаторы класса хранения являются частью синтаксиса объявления имени. Вместе с областью видимости имени они управляют двумя независимыми свойствами имени: продолжительностью хранения (`storage duration`) и связыванием (`linkage`).

- `auto` (устаревшее). Начиная с C++11 означает другое;
- `register` (устаревшее). Не используется начиная с C++17;
- `static`;
- `extern`;
- `thread_local`.

// https://en.cppreference.com/w/cpp/language/storage_duration.html

// Подробнее в практической работе 2

Составные типы

Пререквизит: модель памяти программы



Принадлежит программе с момента запуска.
Средний размер: 2 – 8 МБ

Не принадлежит программе, но можно попросить у ОС.
Вся оставшаяся свободная оперативная память

Статические массивы

Объявление статического массива

```
/* тип */ name[/* размер */];
```

// Примеры объявления

```
int scores[5];           // Массив из 5 целых чисел (не инициализирован)
```

```
double temperatures[24]; // Массив из 24 чисел с плавающей точкой
```

```
char name[50];           // Массив из 50 символов
```

// Память под статические массивы выделяется на стеке, поэтому

// большой статический массив может привести к падению программы.

// Например для FullHD картинки нужно: char name[1920*1080*3] ~ 5.9МБ.

Массивы переменной длины (VLA) - не в стандарте C++

```
// Это НЕ стандартный C++ (работает только в некоторых компиляторах)
```

```
int size;
```

```
std::cin >> size;
```

```
int vla[size]; // Ошибка в стандартном C++!
```

```
// Размер статического массива должен быть известен на этапе компиляции
```

```
int array[42];
```

```
const int size = 42;
```

```
int array[size];
```

```
constexpr int size = 42;
```

```
int array[size];
```

```
int n;
```

```
std::cin >> n;
```

```
const int size = n;
```

```
int array[size]; // Ошибка. Это всё равно VLA
```

Инициализация статических массивов

// Полная инициализация

```
int numbers[5] = {1, 2, 3, 4, 5};
```

// Частичная инициализация (остальные элементы - zero value)

```
int partial[5] = {1, 2};           // [1, 2, 0, 0, 0]
```

// Автоматическое определение размера

```
int auto_size[] = {1, 2, 3, 4}; // Размер = 4
```

// Инициализация нулями

```
int zeros[5] = {};                // [0, 0, 0, 0, 0]
```

```
int zeros2[5] = {0};              // [0, 0, 0, 0, 0]
```

// Uniform initialization (C++11)

```
int modern[] {1, 2, 3, 4, 5};    // Без знака =
```

Доступ к элементам статического массива

```
int arr[5] = {10, 20, 30, 40, 50};

// Чтение значений
std::cout << arr[0];    // 10 (первый элемент)
std::cout << arr[4];    // 50 (последний элемент)

// Запись значений
arr[1] = 25;            // Теперь arr[1] = 25
arr[3] = arr[2] * 2;    // arr[3] = 60

// Выход за границы массива
// arr[5] = 100;        // Неопределенное поведение
// arr[-1] = 100;       // Неопределенное поведение

// Копирование массивов
int a[3] {};
int b[3] {};

a = b;    // Ошибка
```

Сравнение статических массивов

```
int a[3]{1, 2, 3};  
int b[3]{1, 2, 3};  
int c[5]{1, 2, 3, 4, 5};
```

// Сравняются не массивы а их адреса в памяти

```
cout << (a == b) << '\n';  
cout << (a > b) << '\n';  
cout << (a < b) << '\n';  
cout << (a == c) << '\n';
```

Определение размера статического массива

```
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
// Способ 1: sizeof (работает только в той же области видимости)
```

```
size_t size = sizeof(arr) / sizeof(arr[0]); // 10
```

```
// Способ 2: шаблонная функция (C++11)
```

```
template<typename T, size_t N>
```

```
constexpr size_t array_size(T (&)[N]) { return N; }
```

```
size_t size2 = array_size(arr); // 10
```

Обход статического массива

```
int numbers[] = {10, 20, 30, 40, 50};
size_t count = sizeof(numbers) / sizeof(numbers[0]);

// Классический for
for (size_t i = 0; i < count; ++i) {
    std::cout << numbers[i] << " ";
}

// Range-based for (C++11) - ПРЕДПОЧТИТЕЛЬНО
for (int element : numbers) {
    std::cout << element << " ";
}

// Range-based for с ссылкой (для изменения)
for (int& element : numbers) {
    element *= 2; // Удваиваем каждый элемент
}

// Через указатели
for (int* ptr = numbers; ptr != numbers + count; ++ptr) {
    std::cout << *ptr << " ";
}
```

Передача статического массива в функцию

```
// НЕПРАВИЛЬНО - size здесь бесполезен
void print_array(int arr[5]) {
    // sizeof(arr) вернет sizeof(int*), а не размер массива!
}
```

```
// ПРАВИЛЬНО - передаем размер явно
void print_array(const int arr[], size_t size) {
    for (size_t i = 0; i < size; ++i) {
        std::cout << arr[i] << " ";
    }
}
```

```
// Использование
int data[] = {1, 2, 3, 4, 5};
print_array(data, sizeof(data)/sizeof(data[0]));
```


Передача статического массива в функцию

```
void print_array(int arr[5], size_t size); // Одно и то же  
void print_array(int arr[], size_t size); // Одно и то же  
void print_array(int *arr, size_t size); // Одно и то же
```

// Определение

```
void print_array(int arr[], size_t size) {  
    for (size_t i = 0; i < size; ++i) {  
        std::cout << arr[i] << " ";  
    }  
}
```

// Использование

```
int data[] = {1, 2, 3, 4, 5};  
print_array(data, sizeof(data)/sizeof(data[0]));
```

Передача статического массива в функцию

```
// При передаче статического массива в функцию происходит "decay" (деградация)
// массива в указатель на первый элемент этого массива, поэтому
// указывать размер массива в объявлении/определении бесполезно
// Для многомерных массивов это касается первой размерности

// Размер здесь бесполезен
void print_array(int arr[5]) {
    // sizeof(arr) вернет sizeof(int*), а не размер массива!
}

void print_array2D(int arr[][7]) {
}
```

Передача статического массива в функцию по ссылке

```
// Принимаем статический массив по ссылке
void print_array(int (&arr)[5]) {
    // Печатаем
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    int b[6] = {1, 2, 3, 4, 5, 6};

    print_array(a); // ОК
    print_array(b); // Ошибка
}

// В общем случае, не имеет особого смысла
```

Передача статического массива в функцию по указателю

```
// Принимаем статический массив по ссылке
void print_array(int (*arr)[5]) {
    // Печатаем
}
```

```
int main() {
    int a[5] = {1, 2, 3, 4, 5};
    int b[6] = {1, 2, 3, 4, 5, 6};

    print_array(&a); // ОК
    print_array(&b); // Ошибка
}
```

```
// В общем случае, не имеет особого смысла
```

Передача статического массива в функцию

```
// Массив известного размера
template<size_t N>
void process_array(int (&arr)[N]) {
    // Размер N известен внутри функции!
    for (size_t i = 0; i < N; ++i) {
        arr[i] *= 2;
    }
}

// Для многомерных массивов
void print_matrix(int matrix[][4], size_t rows) {
    for (size_t i = 0; i < rows; ++i) {
        for (size_t j = 0; j < 4; ++j) {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << "\n";
    }
}
```

C-строки (символьные массивы)

```
// Строка с завершающим нулем
char name1[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
char name2[6] = "Hello"; // Автоматически добавляет '\0'
char name3[] = "Hello"; // Размер = 6
const char* name4 = "Hello";
```

```
// Функции для работы со строками
#include <cstring>
```

```
char str1[20] = "Hello";
char str2[20] = "World";
```

```
std::strcpy(str1, str2); // Копирование
std::strcat(str1, "!"); // Конкатенация
size_t len = std::strlen(str1); // Длина строки
int cmp = std::strcmp(str1, str2); // Сравнение
```

Современные альтернативы: std::array (C++11)

```
#include <array>

// Объявление
std::array<int, 5> arr = {1, 2, 3, 4, 5};

// Преимущества перед встроенными массивами:
std::cout << arr.size();      // Знает свой размер!
arr = {6, 7, 8, 9, 10};      // Можно присваивать!
std::sort(arr.begin(), arr.end()); // Совместим с STL

// Безопасный доступ
std::cout << arr[2];          // Быстрый доступ
std::cout << arr.at(2);       // С проверкой границ (бросает исключение)

// Range-based for
for (int x : arr) {
    std::cout << x << " ";
}
```

Динамические массивы

// Выделение в куче

```
size_t size = 100;
```

```
int* dynamic_array = new int[size];
```

// Использование

```
for (size_t i = 0; i < size; ++i) {
```

```
    dynamic_array[i] = i;
```

```
}
```

// ОБЯЗАТЕЛЬНО освободить память!

```
delete[] dynamic_array;
```


Современные альтернативы: std::vector

```
#include <vector>
```

```
// Динамический массив
```

```
std::vector<int> vec = {1, 2, 3, 4, 5};
```

```
// Можно изменять размер
```

```
vec.push_back(6);           // Добавить элемент
```

```
vec.pop_back();             // Удалить последний элемент
```

```
vec.resize(10);             // Изменить размер
```

```
// Все преимущества std::array +
```

```
// автоматическое управление памятью
```

Многомерные статические массивы

// Двумерный массив (матрица 3x4)

```
int matrix[3][4] = {  
    {1, 2, 3, 4},  
    {5, 6, 7, 8},  
    {9, 10, 11, 12}  
};
```

// Трехмерный массив (куб 2x3x4)

```
int cube[2][3][4] = {  
    {  
        {1,2,3,4}, {5,6,7,8}, {9,10,11,12}  
    },  
    {  
        {13,14,15,16}, {17,18,19,20}, {21,22,23,24}  
    }  
};
```

// Доступ к элементам

```
int value = matrix[1][2];    // 7  
matrix[0][0] = 100;         // Изменяем первый элемент
```

Размещение многомерного статического массива в памяти

// Двумерный массив (матрица 3x4)

```
int matrix[3][4] = {  
    {1, 2, 3, 4},  
    {5, 6, 7, 8},  
    {9, 10, 11, 12}  
};
```

// Элементы расположены в памяти

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12  
matrix[2][3] == matrix[0][11]; // true
```

// Трехмерный массив (куб 2x3x4)

```
int cube[2][3][4] = {  
    {  
        {1,2,3,4}, {5,6,7,8}, {9,10,11,12}  
    },  
    {  
        {13,14,15,16}, {17,18,19,20}, {21,22,23,24}  
    }  
};
```

// Элементы расположены в памяти

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24
```

Многомерные динамические массивы (вариант 1)

```
int rows, cols;
std::cout << "Enter rows and columns: ";
std::cin >> rows >> cols;

// Выделяем память для массива указателей (строк)
int** matrix = new int*[rows];

// Для каждой строки выделяем массив столбцов
for (int i = 0; i < rows; ++i) {
    matrix[i] = new int[cols];
}

// Инициализация значениями
int counter = 1;
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
        matrix[i][j] = counter++;
    }
}

// ОЧЕНЬ ВАЖНО: освобождение памяти в обратном порядке
for (int i = 0; i < rows; ++i) {
    delete[] matrix[i]; // Сначала удаляем каждую строку
}
delete[] matrix;        // Затем массив указателей
```

Многомерные динамические массивы (вариант 2)

```
int rows, cols;
std::cout << "Enter rows and columns: ";
std::cin >> rows >> cols;

// Выделяем буфер
int* buffer = new int[rows*cols];

// Выделяем память для массива указателей (строк)
int** matrix = new int*[rows];

// Устанавливаем указатели на начало строк
for (int i = 0; i < rows; ++i) {
    matrix[i] = &buffer[i*cols];
}

// Инициализация значениями
int counter = 1;
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
        matrix[i][j] = counter++;
    }
}

delete[] matrix;          // Удаляем массив указателей
delete[] buffer;          // Затем удаляем буфер
```

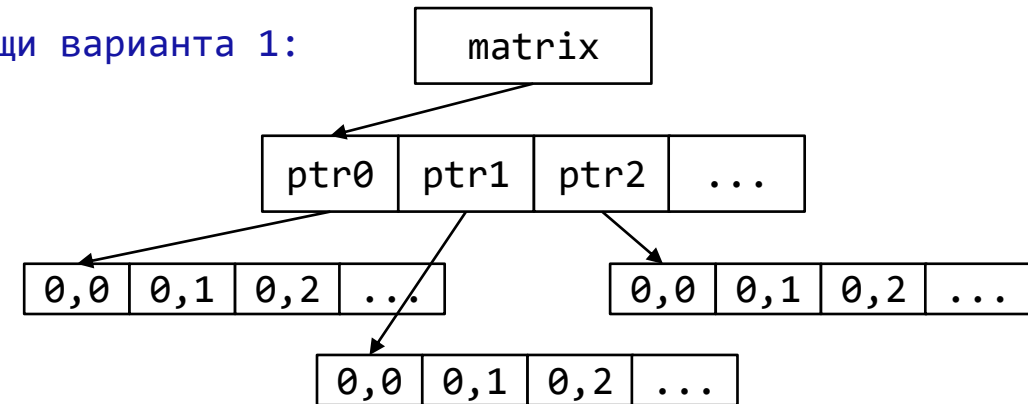
Использование многомерного динамического массива

```
// Использование массива
std::cout << "Matrix[" << rows << "][" << cols << "]:\n";
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
        std::cout << matrix[i][j] << "\t";
    }
    std::cout << "\n";
}
```

Размещение многомерного динамического массива в памяти

При создании многомерного динамического массива при помощи варианта 1:

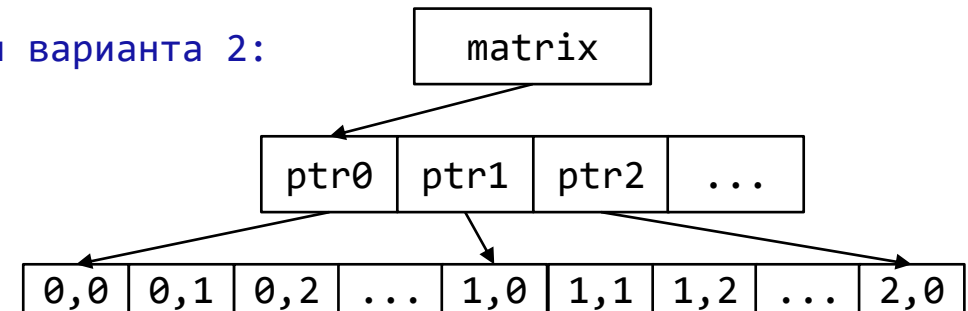
matrix → [ptr0] → [0,0][0,1][0,2]... // Первая строка
[ptr1] → [1,0][1,1][1,2]... // Вторая строка
[ptr2] → [2,0][2,1][2,2]... // Третья строка
...



При этом каждая строка может располагаться в памяти где угодно.

При создании многомерного динамического массива при помощи варианта 2:

matrix → [ptr0] → [0,0][0,1][0,2]... // Первая строка
[ptr1] → [1,0][1,1][1,2]... // Вторая строка
[ptr2] → [2,0][2,1][2,2]... // Третья строка
...



При этом строки располагаются в памяти друг за другом, как и в случае статического массива.

Создание 3D динамического массива

```
int x, y, z;
std::cout << "Enter dimensions (x y z): ";
std::cin >> x >> y >> z;

// Выделение памяти для 3D массива
int*** cube = new int**[x];
for (int i = 0; i < x; ++i) {
    cube[i] = new int*[y];
    for (int j = 0; j < y; ++j) {
        cube[i][j] = new int[z];
    }
}

// Инициализация
int counter = 1;
for (int i = 0; i < x; ++i) {
    for (int j = 0; j < y; ++j) {
        for (int k = 0; k < z; ++k) {
            cube[i][j][k] = counter++;
        }
    }
}

// Освобождение памяти (в обратном порядке!)
for (int i = 0; i < x; ++i) {
    for (int j = 0; j < y; ++j) {
        delete[] cube[i][j]; // Удаляем каждый "столбец" Z
    }
    delete[] cube[i];       // Удаляем каждый "слой" Y
}
delete[] cube;              // Удаляем основной массив
```


Современные альтернативы - вектор векторов

```
size_t rows, cols;
std::cout << "Enter rows and columns: ";
std::cin >> rows >> cols;

// Создание матрицы как вектора векторов
std::vector<std::vector<int>> matrix(rows, std::vector<int>(cols));

// Инициализация
int counter = 1;
for (auto& row : matrix) {
    for (auto& element : row) {
        element = counter++;
    }
}

// Использование
for (const auto& row : matrix) {
    for (const auto& element : row) {
        std::cout << element << "\t";
    }
    std::cout << "\n";
}
```

Структуры

Постановка задачи

- Хранить в программе описание характеристик некоторого объекта

Решение I

```
int aliceBirthYear;  
int aliceBirthMonth;  
int aliceBirthDay;  
double aliceHeight;  
double aliceWeight;
```

```
int bobBirthYear;  
int bobBirthMonth;  
int bobBirthDay;  
double bobHeight;  
double bobWeight;
```

Решение I - Проблемы

- Для каждого человека нужно создавать по пять отдельных переменных – **долго, могут быть опечатки**
- Чтобы передать в функцию, нужно перечислить все аргументы – **можно перепутать порядок**

```
print(aliceBirthYear, aliceBirthMonth,  
      aliceBirthDay, aliceHeight, aliceWeight);
```

- Как вернуть из функции?

Решение II - Структуры

```
struct human {      // Свой тип данных
    int BirthYear;
    int BirthMonth;
    int BirthDay;
    double Height;
    double Weight;
};    // Точка с запятой обязательно

human alice, bob;    // Создаём переменные
```

Решение II – Структуры

```
struct human {  
    int BirthYear;  
    int BirthMonth;  
    int BirthDay;  
    double Height;  
    double Weight;  
} alice, bob;
```

Решение II – Анонимные структуры

```
struct {  
    int BirthYear;  
    int BirthMonth;  
    int BirthDay;  
    double Height;  
    double Weight;  
} alice, bob;
```


Где можно объявлять структуры?

- Внутри функций

```
void func(){
    struct num{int i;} var;
};
```
- Вне функций

```
struct num{int i;} var;
void func(){
};
```
- Внутри других структур

```
struct num{
    int i;
    struct {int k;} j;
} var;
```

Что может быть членом структуры?

Если можно создать переменную этого типа, то это может быть членом структуры

Например:

- Примитивные типы: `int`, `double`, `char` ...
- Другие структуры;
- Массивы;
- Строки;
- ...

Как работать со структурой

```
struct Data{  
    int Year;  
    int Month;  
    int Day;  
};
```

```
Data now;  
now.Year = 2018;  
now.Day = 9;  
now.Month = 11;
```

Как работать со структурой

```
now.Year = now.Year + 1; // 2019
```

```
cout << now.Day; // 9
```

```
now.Month = now.Day + now.Year; // 2028
```

```
int *p = &now.Month;
```

Инициализация структуры I

```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
// joe.id = 1, joe.age = 32, joe.wage = 60000.0
```

```
Employee joe = { 1, 32, 60000.0 };
```

```
// frank.id = 2, frank.age = 28, frank.wage = 0.0
```

```
Employee frank = { 2, 28 };
```

```
Employee frank { 2, 28 };    // C++11
```

Инициализация структуры I - Вложенные структуры

```
struct Address {  
    string street;  
    string city;  
};
```

```
struct Employee {  
    short id;  
    Address address;  
    string name;  
};
```

```
Employee john {1, {"123 Main St", "Springfield"}, "John"};
```

Инициализация структуры II C++11/C++14

```
struct Rectangle {  
    double length = 1.0;  
    double width = 1.0;  
};
```

```
int main() {  
    Rectangle x; // length = 1.0, width = 1.0  
    x.length = 2.0; // Меняем значение  
    return 0;  
}
```

Инициализация структуры III C++11/C++14

```
struct Rectangle {  
    double length = 1.0;  
    double width = 1.0;  
};  
  
int main() {  
    // C++11 – Ошибка; C++14 – Разрешено  
    Rectangle x = {1.0, 1.0};  
  
    return 0;  
}
```


Присваивание значений структурам I

```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
Employee joe;  
joe.id = 1;  
joe.age = 32;  
joe.wage = 60000.0;
```

Присваивание значений структурам II

```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
Employee joe = {1, 20, 3.0}, mike;  
mike = joe; // Копирование значений joe в mike  
  
// Присваивание полям joe новых значений C++14  
joe = {2, 22, 6.3};
```

Присваивание значений структурам III - Designated Initializers (Именованные инициализаторы) в C++20

```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
Employee joe{.id = 1, .age = 20, .wage = 6.5};
```

```
Employee joe{.id = 1, .wage = 6.5}; // age = 0
```

```
Employee joe{.age = 20, .id = 1, .wage = 6.5}; // Ошибка. Порядок важен
```

```
Employee joe{1, .age = 20, .wage = 6.5}; // Ошибка. Смешанная инициализация
```

Присваивание значений структурам III - Designated Initializers (Именованные инициализаторы) в C++20

```
struct Address {
    string street;
    string city;
};

struct Employee {
    short id;
    string name;
    Address address;
};

// Инициализация вложенных структур
Employee john {
    .id = 1,
    .name = "John",
    .address = {
        .street = "123 Main St",
        .city = "Springfield",
    }
};

// Можно комбинировать с обычной инициализацией для вложенных структур
Employee john {
    .id = 1,
    .name = "John",
    .address = {"123 Main St", "Springfield"}
};
```

Передача структуры как параметр в функцию

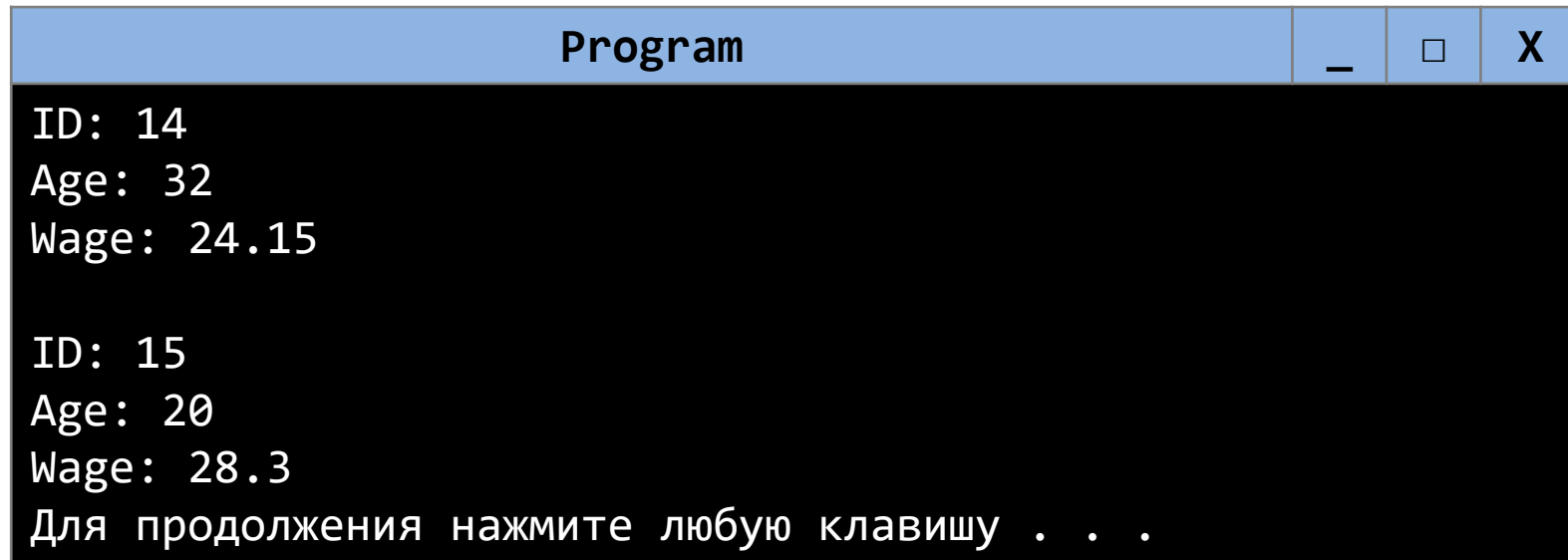
```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
void printInformation(Employee employee) {  
    std::cout << "ID: " << employee.id << "\n";  
    std::cout << "Age: " << employee.age << "\n";  
    std::cout << "Wage: " << employee.wage << "\n";  
}
```

Передача структуры как параметр в функцию

```
int main() {  
    Employee joe = { 14, 32, 24.15 };  
  
    printInformation(joe);  
    std::cout << "\n";  
  
    printInformation({ 15, 20, 28.3 });  
    return 0;  
}
```

Передача структуры как параметр в функцию



The screenshot shows a standard Windows console window with a light blue title bar labeled 'Program'. The window contains white text on a black background. The text displays two sets of data: the first set shows 'ID: 14', 'Age: 32', and 'Wage: 24.15'; the second set shows 'ID: 15', 'Age: 20', and 'Wage: 28.3'. At the bottom, there is a prompt in Russian: 'Для продолжения нажмите любую клавишу . . .', which translates to 'Press any key to continue . . .'. The window has standard minimize, maximize, and close buttons on the right side of the title bar.

```
Program
ID: 14
Age: 32
Wage: 24.15

ID: 15
Age: 20
Wage: 28.3
Для продолжения нажмите любую клавишу . . .
```

Передача структуры в функцию через указатель

```
void printInformation(Employee *employee) {  
    std::cout << "ID: " << (*employee).id << "\n";  
    std::cout << "Age: " << (*employee).age << "\n";  
    std::cout << "Wage: " << (*employee).wage << "\n";  
}
```

```
void printInformation(Employee *employee) {  
    std::cout << "ID: " << employee->id << "\n";  
    std::cout << "Age: " << employee->age << "\n";  
    std::cout << "Wage: " << employee->wage << "\n";  
}
```


Возврат структур из функций

```
struct Point3d {  
    double x, y, z;  
};
```

```
Point3d getZeroPoint() {  
    Point3d temp = { 0.0, 0.0, 0.0 };  
    return temp;  
}
```

```
int main() {  
    Point3d zero = getZeroPoint();  
    return 0;  
}
```

Дополнительные сведения

Разные типы

```
struct Point3d {  
    double x, y, z;  
};
```

```
struct Vector3d {  
    double x, y, z;  
};
```

```
Point3d p = { 0.0, 0.0, 0.0 };
```

```
Vector3d v;
```

```
v = p; // Ошибка. У v и p разные типы
```

Массив структур

```
struct Point3d {  
    double x, y, z;  
};
```

```
Point3d p[2] = {{}, {1.0, 2.0, 3.0}};
```

```
p[0].x = 1.0;
```

```
std::cout << p[0].x << ' ' << p[0].y << ' ' << p[0].z;
```

Размер структуры и выравнивание I

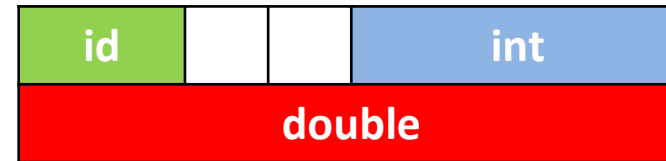
```
struct Employee {  
    short id;    // sizeof(short) == 2  
    int age;     // sizeof(int) == 4  
    double wage; // sizeof(double) == 8  
};  
  
sizeof(Employee); // 16 != ( 2 + 4 + 8 )
```

Размер структуры и выравнивание II

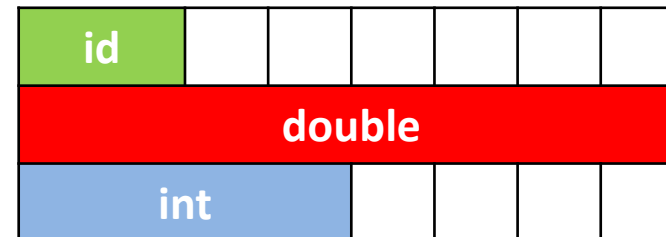
```
struct Employee {  
    short id;      // sizeof(short) == 2  
    double wage;   // sizeof(double) == 8  
    int age;       // sizeof(int) == 4  
};  
  
sizeof(Employee); // 24 != ( 2 + 4 + 8 )
```

Размер структуры и выравнивание II

```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```



```
struct Employee {  
    short id;  
    double wage;  
    int age;  
};
```



Объединения

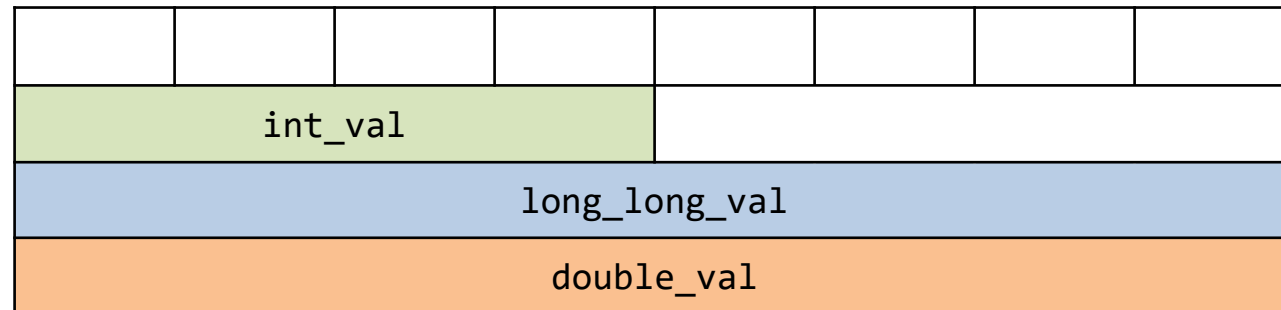
Что такое объединение?

Объединение — это пользовательский тип данных, который может хранить в пределах *одной области* памяти разные типы данных, но в каждый момент времени только один из них.

Размер объединения определяется размером крупнейшего поля.

Объявление

```
union one4all {  
    int int_val;  
    long long long_long_val;  
    double double_val;  
};
```



Инициализация

```
union one4all {  
    int int_val;  
    long long long_long_val;  
    double double_val;  
};
```

// Выражение вычисляется и присваивается
первому полю в объединении

```
one4all num = {10.1};  
cout << num.int_val; // 10
```

Использование I

```
one4all pail;  
pail.int_val = 15;           // сохранение int  
cout << pail.int_val;       // 15  
  
pail.double_val = 1.38;     // сохранение double  
cout << pail.double_val;    // 1.38  
cout << pail.int_val;       // -515396076
```

Использование II

```
struct widget {  
    char brand[20];  
    int type; // Определяет что лежит в id_val  
    union id {  
        long id_num;  
        char id_char[20];  
    } id_val;  
};
```

Анонимные объединения

```
union {                                // Нет имени
    long id_num;
    char id_char[20];
};                                     // Нет переменных
```

Две переменные работающие с одной областью памяти

Анонимные объединения

```
int main(){  
    union {  
        int i;  
        double d;  
    };  
    i = 12345678;  
    cout << i << '\n';    // 12345678  
  
    d = 12345678;  
    cout << i << '\n';    // -1073741824  
}
```

Современная альтернатива - std::variant

```
#include <variant>
```

```
#include <string>
```

```
// Объявление variant-а, который может хранить int, double или std::string  
std::variant<int, double, std::string> myVariant;
```

```
myVariant = 42; // Активен int
```

```
myVariant = 3.14; // Активен double
```

```
myVariant = "Hello, world!"; // Активен std::string
```


Современная альтернатива - std::variant

```
#include <variant>
```

```
#include <string>
```

```
// Объявление variant-а, который может хранить int, double или std::string  
std::variant<int, double, std::string> myVariant;
```

```
// Проверка активного типа
```

```
if (std::holds_alternative<int>(myVariant)) {
```

```
    // Делаем что-то с int
```

```
}
```

Современная альтернатива - std::variant

```
// Способ 1: std::get (бросает исключение при ошибке типа)
```

```
try {  
    int value = std::get<int>(myVariant);  
} catch (const std::bad_variant_access& e) {  
    // Обработка ошибки несовпадения типа  
}
```

```
// Способ 2: std::get_if (возвращает указатель, безопасно)
```

```
if (auto* pval = std::get_if<int>(&myVariant); pval) {  
    // Используем *pval. Если pval == nullptr, он преобразуется в false  
}
```

Современная альтернатива - std::variant (Visitor Pattern)

```
#include <variant> // для std::visit

// Определяем visitor, который обрабатывает все возможные типы
struct MyVisitor {
    void operator()(int i)      { std::cout << "int: " << i << "\n"; }
    void operator()(double d) { std::cout << "double: " << d << "\n"; }
    void operator()(const std::string& s) { std::cout << "string: " << s << "\n"; }
};

std::visit(MyVisitor{}, myVariant); // Применяем visitor
```

Перечисления

Что такое перечисление

Перечисление – это пользовательский тип данных, определяющий набор целочисленных констант.

Зачем нужен:

- Сделать код более читабельным путём замены «магических чисел» на элементы перечисления;
Пример: `return 0;` `return SUCCESS;`
- Как дополнительный контроль, защищающий от случайных, автоматических преобразований типов.

Объявление I

```
enum Color {  
    // Элементы перечисления называются перечислителями  
    // Они определяют все допустимые значения данного типа  
    COLOR_BLACK,    // Перечислители разделяются запятыми  
    COLOR_RED,      // Обычно они пишутся заглавными буквами  
    COLOR_BLUE,     // но это не обязательно  
    COLOR_GREEN,  
    COLOR_WHITE,  
    COLOR_CYAN,  
    COLOR_YELLOW,  
    COLOR_MAGENTA, // В C++11 можно ставить запятую в конце  
};                // Точка с запятой обязательна
```

Объявление II

```
enum Color {  
    COLOR_BLACK,          // Присваивается целое значение 0  
    COLOR_RED,            // 1  
    COLOR_BLUE = 7,       // Можно присвоить своё значение  
    COLOR_GREEN,          // 8 Нумерация продолжается  
    COLOR_WHITE = 7,      // Можно дублировать значения  
    COLOR_CYAN,           // 8  
    COLOR_YELLOW,         // 9  
    COLOR_MAGENTA = -1    // отрицательные тоже допускаются  
};
```

Объявление III

```
enum Color {  
    YELLOW,  
    BLACK, // имя BLACK теперь занято  
    PINK  
};
```

```
enum Feelings {  
    SAD,  
    ANGRY,  
    BLACK // ошибка, BLACK уже использован в Colors  
};
```

```
int BLACK = 3; // ошибка
```


Переменные

```
enum Color {  
    YELLOW,  
    BLACK,  
    PINK  
} a;           // Создание во время объявления
```

```
enum {  
    ONE,  
    TWO,  
    THREE  
} b;           // Создание из анонимного перечисления
```

```
Color c;       // Обычным способом
```

Инициализация / Присваивание

```
enum Color {  
    YELLOW,  
    BLACK,  
    PINK  
};
```

```
Color c = YELLOW;  
Color pig(PINK);  
Color zebr = Color::BLACK;  
Color window = 0;  // Ошибка
```

Ввод / Вывод

```
enum Color {  
    YELLOW,  
    BLACK,  
    PINK  
};
```

```
Color pig(PINK);  
cout << pig;      // Преобразуется в число (2)  
cin >> pig;       // Ошибка компиляции
```

```
int input;  
cin >> input;  
pig = static_cast<Color>(input);
```

Операции I

```
enum Color { YELLOW, BLACK, PINK };
```

Перечисления преобразуются в целое число автоматически:

```
Color c = BLACK;
```

```
int i = 5 + c;      // i = 5 + 1;
```

```
int j = 5 + PINK;   // j = 5 + 2;
```

Переменной перечисляемого типа можно присвоить только перечислитель соответствующего типа:

```
Color dor = YELLOW;
```

```
c = dor;
```

```
c = 0;  // Ошибка    c != YELLOW
```

```
c = static_cast<Color>(0); // Явное преобразование можно
```

Операции II

```
enum Color { YELLOW, BLACK, PINK };  
Color c = BLACK, pig = PINK;
```

Переменные перечисляемого типа часто используются в:

- Операторах ветвления:

```
if (pig == PINK) ...;  
switch(c){  
    case YELLOW : ...; break;  
    case BLACK  : ...; break;  
}
```
- В качестве возвращаемого значения:

```
return ERROR_OPENING_FILE;  
return SUCCESS;
```

Операции III

```
enum Color {  
    YELLOW = 0,  
    BLACK = 1,  
    PINK = 2  
};  
  
// Функция  
void setColor(Color c);  
  
// Использование  
Color pig = PINK;  
setColor(pig);    // ОК  
setColor(YELLOW); // ОК  
setColor(0);      // Ошибка
```

Явное преобразование перечислений

```
enum Color { YELLOW, BLACK, PINK = 10 };
```

```
Color c = BLACK;
```

```
c = static_cast<Color>(0);
```

```
c = Color(0);    // в стиле Си
```

```
c = (Color) 0;   // в стиле Си
```

// Допускается, но поведение будет не определено

```
c = static_cast<Color>(5);
```

Альтернатива - enum class

// Базовый синтаксис

```
enum class Имя [ : базовый_тип ] { список_перечислителей };
```

// Примеры

```
enum class Color { RED, GREEN, BLUE }; // Базовый тип по умолчанию - int
```

```
enum class Priority : uint8_t { LOW, MEDIUM, HIGH }; // Явный базовый тип
```

```
enum class Permissions : uint8_t {  
    NONE = 0,  
    READ = 1 << 0,    // 1  
    WRITE = 1 << 1,   // 2  
    EXECUTE = 1 << 2, // 4  
    ALL = READ | WRITE | EXECUTE // 7  
};
```


Альтернатива - enum class

```
enum class Color { RED, GREEN, BLUE };
enum class TrafficLight { RED, YELLOW, GREEN }; // OK - нет конфликта!

int main() {
    Color c = Color::RED; // Обязательно квалифицированное имя
    // Color c2 = RED;    // Ошибка
    TrafficLight tr = TrafficLight::RED;

    int value = c; // Ошибка
    cout << c;     // Ошибка
    cin >> c;      // Ошибка

    int value = static_cast<int>(c); // OK. Явное преобразование
    c = static_cast<Color>(0);       // OK. Явное преобразование
}
```

Информация о типах

Оператор sizeof - размер типа в байтах

```
// Размеры фундаментальных типов
cout << "sizeof(int): " << sizeof(int) << " bytes\n";
cout << "sizeof(double): " << sizeof(double) << " bytes\n";
cout << "sizeof(char): " << sizeof(char) << " bytes\n";

// Размеры выражений и переменных
int x = 42;
cout << "sizeof(x): " << sizeof(x) << " bytes\n";
cout << "sizeof(x + 1.0): " << sizeof(x + 1.0) << " bytes\n";

// Размеры структур
struct Point { double x, y, z; };
cout << "sizeof(Point): " << sizeof(Point) << " bytes\n";

// Размеры массивов
int arr[10];
cout << "sizeof(arr): " << sizeof(arr) << " bytes\n";
cout << "Number of elements: " << sizeof(arr) / sizeof(arr[0]) << "\n";
```

Оператор alignas (C++11) – выравнивание в байтах

```
struct Struct {  
    int a;  
    double b;  
};
```

// Смотрим требование к выравниванию

```
cout << "alignof(int): " << alignof(int) << "\n";           // 4  
cout << "alignof(double): " << alignof(double) << "\n";       // 8  
cout << "alignof(Struct): " << alignof(Struct) << "\n";       // 8
```

Оператор alignas (C++11) – выравнивание в байтах

```
struct alignas(16) AlignedStruct {  
    int a;  
    double b;  
};
```

// Требуем выравнивать int кратно 8 байтам

```
alignas(8) int aligned_var;
```

```
cout << "alignof(aligned_var): " << alignof(aligned_var) << "\n";    // 8
```

```
cout << "alignof(AlignedStruct): " << alignof(AlignedStruct) << "\n"; // 16
```

Оператор typeid и RTTI (Runtime Type Information)

```
#include <iostream>
#include <typeinfo> // для typeid

int main() {
    int i = 42;
    double d = 3.14;
    std::string s = "hello";

    // Получение информации о типах
    cout << "Type of i: " << typeid(i).name() << "\n"; // Type of i: i
    cout << "Type of d: " << typeid(d).name() << "\n"; // Type of d: d
    cout << "Type of s: " << typeid(s).name() << "\n"; // Type of s: NSt7__cxx1112basic_string

    // Сравнение типов
    if (typeid(i) == typeid(int)) {
        cout << "i это int\n"; // i это int
    }
}

// https://en.cppreference.com/w/cpp/utility/rtti.html
```

type_traits (C++17)

```
#include <iostream>
#include <type_traits>

template<typename T>
void check_type() {
    std::cout << "is_const: " << std::is_const<T>::value << "\n";
    std::cout << "is_volatile: " << std::is_volatile<T>::value << "\n";
    std::cout << "is_signed: " << std::is_signed<T>::value << "\n";
    std::cout << "is_unsigned: " << std::is_unsigned<T>::value << "\n";
    std::cout << "is_floating_point: " << std::is_floating_point<T>::value << "\n";
}

int main() {
    check_type<const int>();
    check_type<volatile double>();
    check_type<unsigned long>();
}

// https://en.cppreference.com/w/cpp/header/type\_traits.html
```

type_traits использование (C++17)

```
#include <iostream>
#include <type_traits>

template<typename T>
void process(const T& value) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Integral: " << value * 2 << "\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Floating: " << value / 2.0 << "\n";
    } else if constexpr (std::is_pointer_v<T>) {
        std::cout << "Pointer: " << *value << "\n";
    } else {
        std::cout << "Other type\n";
    }
}

int main() {
    process(42);           // Integral
    process(3.14);         // Floating
    int x = 100;
    process(&x);           // Pointer
    process("hello");      // Other type
}
```


Концепты (C++20)

```
#include <iostream>
#include <concepts>
#include <string>

// Концепт
template<typename T>
concept Arithmetic = std::integral<T> || std::floating_point<T>;

// Используем концепт как ограничение на тип
template<Arithmetic T>
T square(T value) {
    return value * value;
}

int main() {
    std::cout << "Square of 5: " << square(5) << "\n";    // OK
    std::cout << "Square of 2.5: " << square(2.5) << "\n"; // OK
    std::string str = "10";
    std::cout << "Square of str: " << square(str) << "\n"; // Ошибка
}
```

Концепты (C++20)

```
#include <iostream>
#include <concepts>
#include <vector>

// Концепты требуют наличия метода и определённого типа результата
template<typename T>
concept HasSize = requires(T t) {
    t.size();
    { t.size() } -> std::convertible_to<size_t>;
};

// Используем концепт как ограничение на тип
template<HasSize Container>
void print_size(const Container& container) {
    std::cout << "Size: " << container.size() << "\n";
}

int main() {
    std::vector<int> vec{1, 2, 3};
    print_size(vec);    // OK
    // print_size(42); // Ошибка
}

// https://en.cppreference.com/w/cpp/concepts.html
```

Информация о стандартных числовых типах

```
#include <iostream>
#include <limits>

int main()
{
    std::cout << "type\t| lowest()\t| min()\t\t| max()\n"
               << "bool\t| "
               << std::numeric_limits<bool>::lowest() << "\t\t| "
               << std::numeric_limits<bool>::min() << "\t\t| "
               << std::numeric_limits<bool>::max() << '\n'
               << "uchar\t| "
               << +std::numeric_limits<unsigned char>::lowest() << "\t\t| "
               << +std::numeric_limits<unsigned char>::min() << "\t\t| "
               << +std::numeric_limits<unsigned char>::max() << '\n'
               << "int\t| "
               << std::numeric_limits<int>::lowest() << "\t| "
               << std::numeric_limits<int>::min() << "\t| "
               << std::numeric_limits<int>::max() << '\n'
               << "double\t| "
               << std::numeric_limits<double>::lowest() << "\t| "
               << std::numeric_limits<double>::min() << "\t| "
               << std::numeric_limits<double>::max() << '\n';
}
```

// https://en.cppreference.com/w/cpp/types/numeric_limits.html

decltype - получение типа выражения

```
#include <iostream>
#include <vector>

int main() {
    int x = 42;
    double y = 3.14;

    // Получение типа переменной
    decltype(x) x_copy = x; // int
    decltype(y) y_copy = y; // double

    // Получение типа выражения
    decltype(x + y) result = x + y; // double

    // Тип возвращаемого значения функции
    std::vector<int> vec{1, 2, 3};
    decltype(vec.size()) size = vec.size(); // std::size_t

    std::cout << "Type of result: " << typeid(result).name() << "\n";
    std::cout << "Type of size: " << typeid(size).name() << "\n";
}
```

decltype с auto в функциях (C++14)

```
#include <iostream>

// Автоматический вывод возвращаемого типа
template<typename T, typename U>
auto add(T a, U b) -> decltype(a + b) {
    return a + b;
}

// C++14: упрощенный синтаксис
template<typename T, typename U>
auto multiply(T a, U b) {
    return a * b;
}

int main() {
    auto result1 = add(5, 3.14);    // double
    auto result2 = multiply(2, 4.5); // double
}
```

std::declval для получения фиктивного объекта

```
#include <iostream>
#include <utility>
#include <type_traits>

class NonConstructible {
private:
    NonConstructible() = delete;
public:
    double compute() const { return 3.14; }
};

template<typename T>
void check_compute() {
    // Используем declval для получения "фиктивного" объекта
    using ResultType = decltype(std::declval<T>().compute());

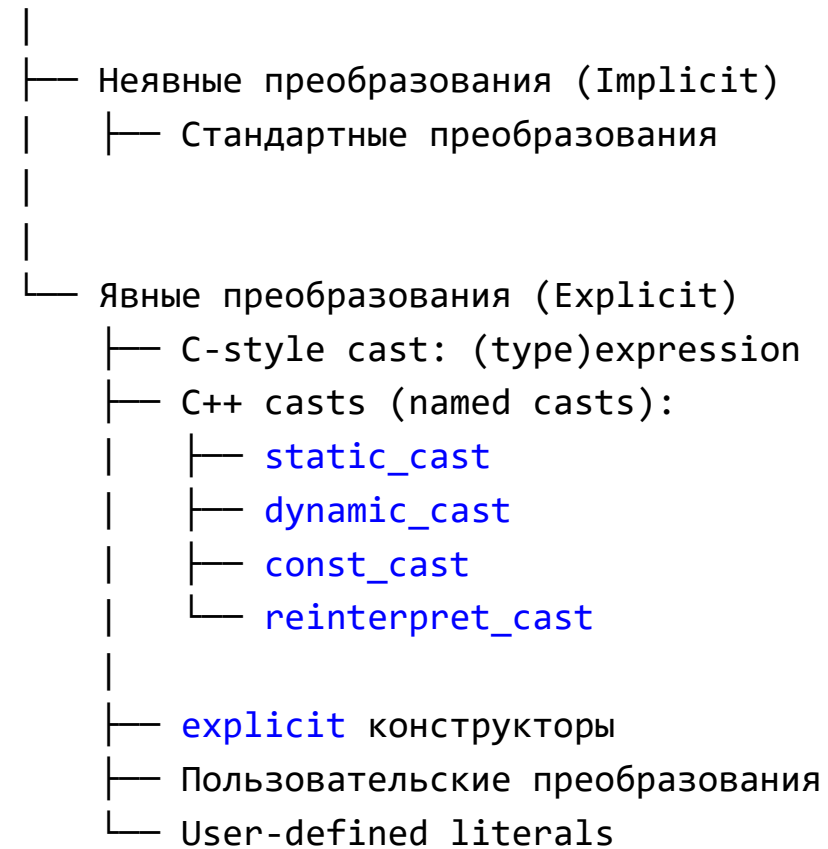
    static_assert(std::is_same<ResultType, double>::value,
                  "compute() should return double");
}

int main() {
    check_compute<NonConstructible>();
    std::cout << "NonConstructible::compute() returns double\n";
}
```

Преобразование типов

Преобразование типов

Преобразования типов в C++



Неявные преобразования (Implicit Conversions)

```
#include <iostream>
#include <string>

class StringWrapper {
private:
    std::string data;

public:
    // Конструктор преобразования
    StringWrapper(const int a) : data(std::to_string(a)) {}

    // Оператор преобразования
    operator std::string() const { return data; }
    operator const char*() const { return data.c_str(); }
};

void processString(const std::string& str) {
    std::cout << "Processing: " << str << "\n";
}

int main() {
    StringWrapper wrapper = 1; // int ->StringWrapper

    std::string str = wrapper; // StringWrapper -> std::string
    const char* cstr = wrapper; // StringWrapper -> const char*
}
```

static_cast – если информация от типах известна на этапе компиляции

```
class Base {};  
  
class Derived : public Base {};  
  
class Unrelated {};  
  
// Числовые преобразования  
double d = 3.14159;  
int i = static_cast<int>(d); // Явное преобразование  
std::cout << "Double: " << d << " -> Int: " << i << "\n";  
  
// Преобразования в иерархии наследования  
Derived derived;  
Base* base_ptr = static_cast<Base*>(&derived); // Upcast - безопасно  
  
// Downcast - опасно, но допустимо если уверены  
Base* base_ptr2 = &derived;  
Derived* derived_ptr = static_cast<Derived*>(base_ptr2);  
derived_ptr->specificMethod();  
  
// Преобразования между связанными типами  
void* void_ptr = static_cast<void*>(&i);  
int* int_ptr = static_cast<int*>(void_ptr);  
  
// Ошибка компиляции - несвязанные типы  
// Unrelated* unrelated = static_cast<Unrelated*>(base_ptr);
```

dynamic_cast – если информация о типах известна на этапе исполнения

```
#include <iostream>
#include <typeinfo>

class Base {
public:
    virtual ~Base() = default;
    virtual void identify() { std::cout << "Base\n"; }
};

class Derived : public Base {
public:
    void identify() override { std::cout << "Derived\n"; }
    void specificMethod() { std::cout << "Specific method\n"; }
};

class AnotherDerived : public Base {
public:
    void identify() override { std::cout << "AnotherDerived\n"; }
};

void safe_process(Base* ptr) {
    // Безопасное приведение вниз
    if (Derived* derived = dynamic_cast<Derived*>(ptr)) {
        std::cout << "It's Derived: ";
        derived->specificMethod();
    } else if (AnotherDerived* another = dynamic_cast<AnotherDerived*>(ptr)) {
        std::cout << "It's AnotherDerived\n";
    } else {
        std::cout << "Unknown derived type\n";
    }
}

void unsafe_process(Base* ptr) {
    // Перекрестное приведение (cross-cast)
    if (AnotherDerived* another = dynamic_cast<AnotherDerived*>(ptr)) {
        std::cout << "Cross-cast succeeded\n";
    } else {
        std::cout << "Cross-cast failed\n";
    }
}
```

```
int main() {
    Derived derived;
    AnotherDerived another;
    Base base;

    safe_process(&derived);    // Успех
    safe_process(&another);    // Успех
    safe_process(&base);       // Неудача

    Base* base_ptr = &derived;
    unsafe_process(base_ptr);  // Cross-cast неудача

    // Работа с ссылками (бросает исключение при неудаче)
    try {
        Derived& derived_ref = dynamic_cast<Derived&>(*base_ptr);
        derived_ref.specificMethod();
    } catch (const std::bad_cast& e) {
        std::cout << "Bad cast: " << e.what() << "\n";
    }
}
```

const_cast – работа с константностью

```
void print(char* str) {
    std::cout << "Non-const: " << str << "\n";
}

void print(const char* str) {
    std::cout << "Const: " << str << "\n";
}

// Снятие константности
const int const_value = 100;
const int* const_ptr = &const_value;

// int* mutable_ptr = const_ptr; // Ошибка
int* mutable_ptr = const_cast<int*>(const_ptr);
*mutable_ptr = 200; // Неопределенное поведение! (original object is const)

// Легальное использование - снятие константности с не-const объекта
int value = 100;
const int* pseudo_const = &value;
int* back_to_mutable = const_cast<int*>(pseudo_const);
*back_to_mutable = 200; // Легально - исходный объект не const

// Использование в перегрузках
const char* const_str = "Hello";
char mutable_str[] = "World";

print(const_cast<char*>(const_str)); // Вызов не-const версии
print(mutable_str); // Вызов не-const версии
print(const_cast<const char*>(mutable_str)); // Вызов константной версии
```

reinterpret_cast – низкоуровневые преобразования байт

```
int value = 0x12345678;

// Преобразование указателей между несвязанными типами
int* int_ptr = &value;
char* char_ptr = reinterpret_cast<char*>(int_ptr);

std::cout << "As bytes: ";
for (int i = 0; i < sizeof(int); ++i) {
    std::cout << std::hex << (int)char_ptr[i] << " ";
}
std::cout << "\n";

// Преобразование указателя в целое и обратно
uintptr_t int_value = reinterpret_cast<uintptr_t>(int_ptr);
int* back_to_ptr = reinterpret_cast<int*>(int_value);

// Преобразование между функциями
using Handler = void(*)(int);
using RawHandler = void(*)(void);

Handler handler = [](int x) { std::cout << "Handler: " << x << "\n"; };
RawHandler raw_handler = reinterpret_cast<RawHandler>(handler);

// ОПАСНО - вызывать через reinterpret_cast
reinterpret_cast<Handler>(raw_handler)(42);

// ОПАСНО
double double_value = reinterpret_cast<uintptr_t>(42);
```

Явные преобразования (C-style cast)

```
double pi = 3.14159;
int int_pi = (int)pi; // C-style cast: отбрасывание дробной части
int _int_pi = int(pi); // аналогично C-style cast

std::cout << "Double: " << pi << "\n";
std::cout << "Int: " << int_pi << "\n";
std::cout << "Int: " << _int_pi << "\n";

// Опасные преобразования
int value = 0x12345678;
char* byte_ptr = (char*)&value; // int* -> char*

// Может делать несколько преобразований сразу
const int const_value = 100;
int* mutable_ptr = (int*)&const_value; // const int* -> int*
*mutable_ptr = 200; // Неопределенное поведение!
```

Явные пользовательские преобразования

```
#include <iostream>
#include <string>

class StringWrapper {
private:
    std::string data;

public:
    // Конструктор преобразования
    explicit StringWrapper(const int a) : data(std::to_string(a)) {}

    // Оператор преобразования
    explicit operator std::string() const { return data; }
    explicit operator const char*() const { return data.c_str(); }
};

int main() {
    // StringWrapper wrapper = 1; // Ошибка
    StringWrapper wrapper2(1);    // OK

    // std::string str = wrapper; // Ошибка
    std::string str2 = static_cast<std::string>(wrapper2); // OK
    // const char* cstr = wrapper; // Ошибка
    const char* cstr2 = static_cast<const char*>(wrapper2); // OK
}
```

User-defined Literals (C++11)

// Вместо громоздкого синтаксиса:

```
Duration d1 = Duration::fromSeconds(5);
```

```
Distance dist1 = Distance::fromMeters(100);
```

// Используем пользовательские литералы:

```
auto d2 = 5_s;           // 5 секунд
```

```
auto dist2 = 100_m;      // 100 метров
```

```
auto weight = 2.5_kg;    // 2.5 килограмм
```

// Общая форма:

```
ReturnType operator"" _suffix(Parameters);
```


User-defined Literals (C++11)

```
constexpr unsigned long long operator"" _kb(unsigned long long bytes) {  
    return bytes * 1024;  
}
```

```
constexpr unsigned long long operator"" _mb(unsigned long long bytes) {  
    return bytes * 1024 * 1024;  
}
```

```
constexpr unsigned long long operator"" _gb(unsigned long long bytes) {  
    return bytes * 1024 * 1024 * 1024;  
}
```

```
auto file_size = 4_gb;      // 4 * 1024^3  
auto memory = 512_mb;      // 512 * 1024^2  
auto buffer = 64_kb;       // 64 * 1024
```

```
std::cout << "File size: " << file_size << " bytes\n";  
std::cout << "Memory: " << memory << " bytes\n";  
std::cout << "Buffer: " << buffer << " bytes\n";
```

Типы контейнеров стандартной библиотеки

Полезные ссылки

Шпаргалки по стандартным контейнерам и не только:

- https://hackingcpp.com/cpp/cheat_sheets.html

Описание стандартных контейнеров

- <https://en.cppreference.com/w/cpp/container.html>

Ассоциативные контейнеры

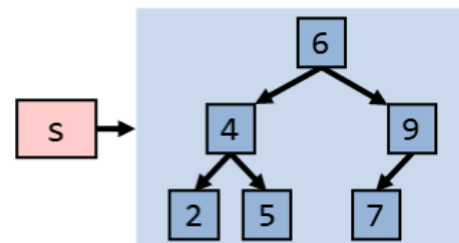
set<Key>

unique, ordered keys

multiset<K>

(non-unique) ordered keys

```
std::set<int> s;  
s.insert(7); ...  
s.insert(5);  
auto i = s.find(7); // → iterator  
if(i != s.end())    // found?  
    cout << *i;    // 7  
if(s.contains(7)) {...} C++20
```



usually implemented
as balanced binary tree
(red-black tree)

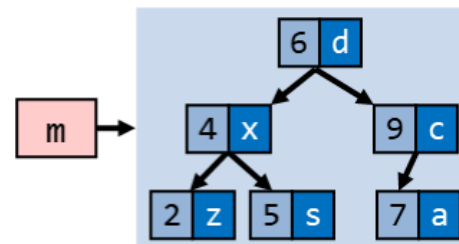
map<Key, Value>

unique key → value-pairs; ordered by keys

multimap<K, V>

(non-unique) key → value-pairs, ordered by keys

```
std::map<int, char> m;  
m.insert({7, 'a'}); ...  
m[4] = 'x'; // insert 4 → x  
auto i = s.find(7); // → iterator  
if(i != s.end())    // found?  
    cout << i->first // 7  
    << i->second; // a  
if(s.contains(7)) {...} C++20
```



usually implemented
as balanced binary tree
(red-black tree)

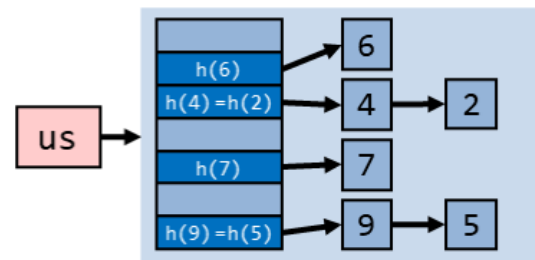
unordered_set<Key>

unique, hashable keys

unordered_multiset<Key>

(non-unique) hashable keys

```
std::unordered_set<int> us;  
us.insert(7); ...  
us.insert(5);  
auto i = us.find(7); // → iterator  
if(i != us.end())    // found?  
    cout << *i;    // 7  
if(s.contains(7)) {...} C++20
```



hash table for
key lookup,
linked nodes
for key storage

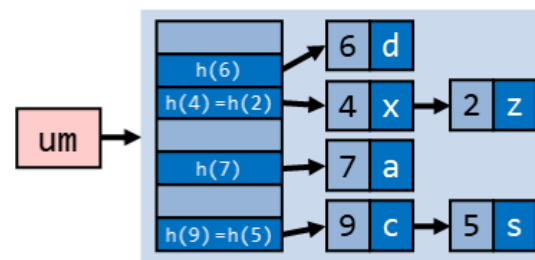
unordered_map<Key, Value>

unique key → value-pairs; hashed by keys

unordered_multimap<Key, Value>

(non-unique) key → value-pairs; hashed by keys

```
unordered_map<int, char> um;  
um.insert({7, 'a'}); ...  
um[4] = 'x'; // insert 4 → x  
auto i = um.find(7); // → iterator  
if(i != um.end())    // found?  
    cout << i->first // 7  
    << i->second; // a  
if(s.contains(7)) {...} C++20
```



hash table for
key lookup,
linked nodes
for (key, value)
pair storage

Множество

Множество

```
#include <iostream>
#include <set>

int main()
{
    // пустое множество чисел int
    std::set<int> numbers;
    // инициализация
    std::set<int> numbers2 {1, 2, 3, 4, 5};
}
```

Размер множества

```
#include <iostream>
#include <set>

int main()
{
    std::set<int> numbers{1, 2, 3};

    std::cout << "Empty: " << std::boolalpha
               << numbers.empty() << std::endl;           // Empty: false
    std::cout << "Size: " << numbers.size() << std::endl; // Size: 3
}
```


Перебор множества

```
#include <iostream>
#include <set>

int main()
{
    std::set<int> numbers{1, 2, 3, 4, 5};

    for (int n : numbers)
        std::cout << n << "\t";
    std::cout << std::endl;
}
```

Добавление элементов

```
#include <iostream>
#include <set>

int main()
{
    std::set<int> numbers{3, 4, 5};

    numbers.insert(1);
    numbers.insert(2);
    numbers.insert(2);
    numbers.insert(2);
    numbers.insert(6);

    for (int n : numbers)
        std::cout << n << " "; // 1 2 3 4 5 6
    std::cout << std::endl;
}
```

Удаление элементов

```
#include <iostream>
#include <set>

int main()
{
    std::set<int> numbers{2, 3, 4, 5};

    numbers.erase(1);
    numbers.erase(2);
    numbers.erase(3);

    for (int n : numbers)
        std::cout << n << " "; // 4 5
    std::cout << std::endl;
}
```

Проверка наличия элемента

```
#include <iostream>
#include <set>

int main()
{
    std::set<int> numbers{2, 3, 4, 5};

    std::cout << "10 in set: " << numbers.count(10) << std::endl; // 10 in set: 0
    std::cout << "2 in set: " << numbers.count(2) << std::endl;    // 2 in set: 1
}
```

Проверка наличия элемента (C++20)

```
#include <iostream>
#include <set>

int main()
{
    std::set<int> numbers{2, 3, 4, 5};

    std::cout << "10 is in set: " << std::boolalpha
               << numbers.contains(10) << std::endl;    // false
    std::cout << "2 is in set: " << std::boolalpha
               << numbers.contains(2) << std::endl;     // true
}
```

Неупорядоченное множество unordered_set

```
#include <iostream>
#include <unordered_set>
#include <set>

int main()
{
    std::set<int> numbers{3, 2, 5, 4};
    numbers.insert(1);
    numbers.insert(6);

    std::unordered_set<int> numbers2{3, 2, 5, 4};
    numbers2.insert(1);
    numbers2.insert(6);

    for (int n : numbers)
        std::cout << n << " "; // 1 2 3 4 5 6
    std::cout << std::endl;

    for (int n : numbers2)
        std::cout << n << " "; // 6 1 4 5 2 3
    std::cout << std::endl;
}
```

std::multiset<KeyType, Compare> (multiple equivalent keys)

Construct A New Set Object

```
set<int> s0 {}  
set<int> s1 {2,1,8,5,4}  
set<int> s2 (begin(s1)+2, end(s1))  
set<int> deep_copy_of_s1 (s1)  
C++17 key type deducible from argument type  
set s3 {7,2,4}; // set<int>
```

Assign New Content To An Existing Set

(deep copy from source)

```
set<int> s1 {1,3,5,7};  
set<int> s2 {4,6,8};  
s1 = s2;  
new state of s1
```

Key Lookup

$O(\log n)$

```
1 3 5 7 .contains(2) → false  
1 3 5 7 .contains(5) → true  
1 3 5 7 .count(2) → 0  
1 3 5 7 .count(5) → 1  
1 3 5 7 .find(2) → @end (= no match)  
1 3 5 7 .find(5) → @match  
1 3 5 7 .lower_bound(3) → @first_not_smaller  
1 3 5 7 .upper_bound(3) → @first_greater  
1 3 5 7 .equal_range(3) → {@lbound, @ubound}
```

can be > 1 only for std::multiset

Query Size (= Number of Keys)

```
2 4 5 .empty() → false  
2 4 5 .size() → 3
```

Erase All Keys

```
2 4 5 .clear()
```

Insert A Single Key

$O(\log n)$

```
2 5 8 .insert(4) → { @inserted, true }  
2 5 8 .insert(5) → { @blocking, false }  
potential performance benefit by hinting at probable insert position  
2 5 8 .insert(@hint, 7) → @inserted_or_block
```

Insert Multiple Keys

$O(\#inserted \cdot \log n)$

```
2 5 8 .insert({1,6,8}) → 1 2 5 6 8  
2 5 8 .insert(@inB, @inE) → 2 3 5 6 8  
source container 3 3 2 6 2
```

Insert & Construct A Key in Place

$O(\log n)$

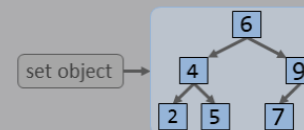
```
set<pair<int,int>> s {{1,3}, {5,6}};  
1,3 5,6 .emplace(4,7) → { @inserted, true }  
potential performance benefit by hinting at probable insert position  
1,3 5,6 .emplace_hint(@hint, 4,7) → @inserted
```

Erase One Key or A Range of Keys

```
1 3 5 7 .erase(2) → 0  
1 3 5 7 .erase(5) → 1  
1 3 5 7 .erase(@pos) → @after_erased  
1 3 5 7 .erase(@beg, @end) → @after_erased
```

$O(\log n)$ #erased
 $O(1)$ amortized
 $O(\log n + \#erased)$

- keys are ordered according to their values
- keys are compared / matched based on *equivalence*:
a and b are equivalent if neither is ordered before the other, e.g., if not (a < b) and not (b < a)
- default ordering comparator is std::less
- sets are usually implemented as a balanced binary tree (e.g., as red-black-tree)



Obtain Iterators

```
0 1 2 .begin() → @first  
0 1 2 .end() → @one_behind_last
```

Obtain Reverse Iterators

```
0 1 2 .rbegin() → rev@last  
0 1 2 .rend() → rev@one_before_first
```

Extract Nodes

C++17

Allows efficient key modification and transfer of keys between different set objects.

```
1 5 7 .extract(5) → 5  
1 5 7 .extract(@pos) → 5
```

$O(\log n)$
 $O(1)$

Merge Two Sets

C++17

```
set<int> s1 {1,3,5,7};  
set<int> s2 {2,5,8};
```

```
1 3 5 7 .merge(2 5 8)  
1 2 3 5 7 8
```

$O(n_2 \cdot \log(n_1 + n_2))$

Insert Nodes

C++17

```
1 7 .insert(5) → { .position | .inserted | .node }  
1 7 .insert(1) → { .position | .inserted | .node }  
1 7 .insert( ) → { .position | .inserted | .node }  
1 7 .insert(@hint, 5) → @inserted  
1 7 .insert(@hint, 1) → @blocking
```

members of the return type
true (empty)
false (empty)
false (empty)
@inserted
@blocking

Modify Key

Direct modification not allowed!
Instead: extract key, modify its value and re-insert.

```
set<int> s {1,5,7};  
auto node = s.extract(5);  
node.value() = 8;  
s.insert(std::move(node));
```

std::unordered_set

Construct A New Set Object

```
unordered_set<int> s0 {}  
unordered_set<int> s1 {2,1,8,4,5}  
unordered_set<int> s2 (begin(s1)+2, end(s1))  
unordered_set<int> deep_copy_of_s1 (s1)  
C++17 key type deducible from argument type  
unordered_set s3 {7,2,4}; // unordered_set<int>
```

Assign New Content To An Existing Set

```
unordered_set<int> s1 {1,5,3,7};  
unordered_set<int> s2 {8,4,6};  
s1 = s2;  
new state of s1  
1 5 3 7 = 8 4 6 → 8 4 6
```

Key Lookup

$O(1)$ average, $O(n)$ worst case

```
1 5 3 7 .contains(2) → false  
1 5 3 7 .contains(3) → true  
1 5 3 7 .count(2) → 0  
1 5 3 7 .count(3) → 1  
can be > 1 only for std::unordered_multiset  
1 5 3 7 .find(2) → @end (= no match)  
1 5 3 7 .find(3) → @match  
1 5 3 7 .equal_range(5) → {@first_equal, @after}
```

Query Size (= Number of Keys)

```
1 5 3 .empty() → false  
1 5 3 .size() → 3
```

Erase All Keys

```
1 5 3 .clear() →
```

Obtain Iterators

(to keys)

```
3 1 5 .begin() → @first  
3 1 5 .end() → @one_behind_last
```

Insert A Single Key

$O(1)$ average, $O(n)$ worst case

```
2 8 5 .insert(4) → {@inserted, true}  
2 8 5 .insert(8) → {@blocking, false}  
potential performance benefit by hinting at probable insert position  
2 8 5 .insert(@hint, 7) → @inserted_or_block  
2 8 7 5
```

Insert Multiple Keys

$O(\#ins)$ avg., $O(n \cdot \#ins + \#ins)$ worst

```
2 8 5 .insert({1,6,8}) → 6 2 5 1 8  
2 8 5 .insert(@inB, @inE) → 2 4 5 3 8  
source container 3 3 2 4 2
```

Insert & Construct A Key in Place

$O(1)$ avg., $O(n)$ worst

```
unordered_set<pair<int,int>> s {{1,3},{5,6}};  
1,3 5,6 .emplace(8,7) → {@inserted, true}  
pair constructor arguments  
potential performance benefit by hinting at probable insert position  
1,3 5,6 .emplace_hint(@hint, 8, 7) → @inserted  
1,3 8,7 5,6
```

Erase One Key or A Range of Keys

$O(\#erased)$ avg., $O(n)$ worst case

```
1 5 3 7 .erase(2) → 0  
1 5 3 7 .erase(3) → 1  
1 5 3 7 .erase(@pos) → @after_erased  
1 5 3 7 .erase(@beg, @end) → @after_erased
```

Query & Control Hash Table Properties

hash function: $h(key) \mapsto \text{bucket index}$

```
unordered_set<char> us {'E','X','Z','A','F','B'};  
US .bucket('E') → 4 (key → hash bucket index)  
4 .begin(4) → @first_in_bucket  
4 .end(4) → @one_behind_last_in_bucket  
2 .bucket_size(4) → 2  
0 .bucket_count() → 7  
us.load_factor() → 4/7 = 0.57  
us.max_load_factor(0.8) (set)  
us.max_load_factor() → 0.8 (get)  
(make table large enough to handle min_capacity elements)  
.reserve(min_capacity) →  
.rehash(#hash_buckets) →  
(can allocate even more buckets if max. load factor demands it)
```

Extract Nodes

$O(1)$ avg., $O(n)$ worst C++17
Allows efficient key modification and transfer of keys between different set objects.

```
1 5 7 .extract(5) → 5  
1 5 7 .extract(@pos) → 5  
1 7
```

Insert Nodes

members of the return type C++17

```
1 7 .insert(8) → {@position|.inserted|.node}  
1 8 7 true (empty)  
1 7 .insert(1) → {@position|.inserted|.node}  
1 7 false 1 (empty node)  
1 7 .insert( ) → {@position|.inserted|.node}  
1 7 false (empty)  
1 7 .insert(@hint, 8) → @inserted  
1 8 7  
1 7 .insert(@hint, 1) → @blocking  
1 7
```

Modify Key

Direct modification not allowed!
Instead: extract key, modify its value and re-insert.

```
unordered_set s {1,7,5};  
auto node = s.extract(5);  
5 1 7  
if (node) { // if key existed  
    node.value() = 8;  
    s.insert(move(node));  
} 8 1 7
```


Словарь

Неупорядоченное множество unordered_set

```
#include <iostream>
#include <map>

int main()
{
    std::map<std::string, unsigned> products;
}
```

Обращение к элементам

```
#include <iostream>
#include <map>

int main()
{
    std::map<std::string, unsigned> products;
    // установка значений
    products["bread"] = 30;
    products["milk"] = 80;
    products["apple"] = 60;

    // получение значений
    std::cout << "bread\t" << products["bread"] << std::endl;
    std::cout << "milk\t" << products["milk"] << std::endl;
    std::cout << "apple\t" << products["apple"] << std::endl;
}
```

Перебор элементов

```
#include <iostream>
#include <map>

int main()
{
    std::map<std::string, unsigned> products;
    // установка значений
    products["bread"] = 30;
    products["milk"] = 80;
    products["apple"] = 60;

    for (const auto& element : products)
        std::cout << element.first << "\t" << element.second << std::endl;
}
```

Перебор элементов (C++17)

```
#include <iostream>
#include <map>

int main()
{
    std::map<std::string, unsigned> products;
    // установка значений
    products["bread"] = 30;
    products["milk"] = 80;
    products["apple"] = 60;

    for (const auto& [product, price] : products)
        std::cout << product << "\t" << price << std::endl;
}
```

Инициализация элементов

```
#include <iostream>
#include <map>

int main()
{
    std::map<std::string, unsigned> products
    {
        std::pair<std::string, unsigned>{"bread", 30},
        std::pair<std::string, unsigned>{"milk", 80},
        std::pair<std::string, unsigned>{"apple", 60}
    };
}
```

Инициализация элементов

```
#include <iostream>
#include <map>

int main()
{
    std::map<std::string, unsigned> products
    {
        {"bread", 30},
        {"milk", 80},
        {"apple", 60}
    };
}
```

Удаление элементов

```
#include <iostream>
#include <map>

int main()
{
    std::map<std::string, unsigned> products
    {
        {"bread", 30}, {"milk", 80}, {"apple", 60}
    };

    products.erase("milk");    // удаляем элемент с ключом "milk"

    for (const auto& [product, price] : products)
        std::cout << product << "\t" << price << std::endl;

    // apple    60
    // bread    30
}
```


Размер словаря

```
#include <iostream>
#include <map>

int main()
{
    std::map<std::string, unsigned> products
    {
        {"bread", 30}, {"milk", 80}, {"apple", 60}
    };

    std::cout << "count: "
                << products.size() << std::endl; // count: 3
    std::cout << "is empty: "
                << std::boolalpha << products.empty() << std::endl; // is empty: false
}
```

Проверка наличия элемента

```
#include <iostream>
#include <map>

int main()
{
    std::map<std::string, unsigned> products
    {
        {"bread", 30}, {"milk", 80}, {"apple", 60}
    };
    std::cout << "Apple\t" << products.count("apple")<< std::endl;    // Apple  1
    std::cout << "Orange\t" << products.count("orange")<< std::endl; // Orange  0

    std::cout << "Apple\t" << std::boolalpha
                << products.contains("apple")<< std::endl;    // Apple  true
    std::cout << "Orange\t" << std::boolalpha
                << products.contains("orange")<< std::endl;    // Orange false
}
```

Неупорядоченные словари

```
#include <iostream>
#include <unordered_map>

int main()
{
    std::unordered_map<std::string, unsigned> products
    {
        {"bread", 30}, {"milk", 80}, {"apple", 60}
    };

    for (const auto& [product, price] : products)
        std::cout << product << "\t" << price << std::endl;
}
```

std::map

Construct A New Map Object

```
map<int,string> m0 { }  
map<int,string> m1 { {4,"Z"},  
                    {2,"A"}, {7,"Y"} }  
map<int,string> m2 (begin(m1)+1,  
                  end(m1))  
map<int,string> deep_copy_of_m1(m1)  
C++17 key and mapped types deducible from arguments  
map m3 { {2,3.14}, {5,6.0} }; // map<int,double>
```

Insert A Single Key-Value Pair

$O(\log n)$

```
1 F 3 A .insert({2,"W"}) -> {@inserted,true}  
1 F 3 A .insert({3,"X"}) -> {@blocking,false}  
1 F 3 A .insert(@hint,{2,"W"}) -> {@ins/block}
```

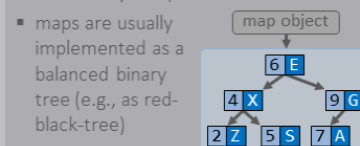
potential performance benefit by hinting at probable insert position

Obtain Iterators

```
1 F 3 A .begin() -> @first  
1 F 3 A .end() -> @one_behind_last  
1 F 3 A .rbegin() -> rev@last  
1 F 3 A .rend() -> rev@one_before_1st
```

$v.begin()$ $v.end()$
 $v.rend()$ $v.rbegin()$
 $.base()$ $.base()$
 $@pos = rev@pos.base() - 1$

- key-value pairs are ordered by key
- key matching is *equivalence*-based:
2 keys *a* and *b* are equivalent if
not (*a* < *b*) and not (*b* < *a*)
- default key comparator is `std::less`



Assign New Content To An Existing Map

```
map<int,string> m1 { {2,"X"} };  
map<int,string> m2 { {1,"A"}, {4,"G"} };  
m1 = m2;  
new state of m1  
2 X = 1 A 4 G -> 1 A 4 G
```

(deep copy from source)

Lookup Using Keys as Input

$O(\log n)$

```
map<int,string> m { {3,"A"}, {5,"X"}, {1,"F"} };  
1 F 3 A 5 X .contains(2) -> false  
1 F 3 A 5 X .contains(5) -> true  
1 F 3 A 5 X .count(2) -> 0  
1 F 3 A 5 X .count(5) -> 1  
1 F 3 A 5 X .find(2) -> @end (=no match)  
1 F 3 A 5 X .find(5) -> @match  
1 F 3 A 5 X .lower_bound(3) -> @1st_not_smaller  
1 F 3 A 5 X .upper_bound(3) -> @1st_greater  
1 F 3 A 5 X .equal_range(3) -> {@Lower,@Upper}  
1 F 3 A 5 X .at(3) -> "A"  
1 F 3 A 5 X .at(2) -> Throws Exception  
std::out_of_range
```

C++20

Query Size (= number of key-value pairs)

```
1 F 3 A .empty() -> false  
1 F 3 A .size() -> 2
```

Insert Multiple Key-Value Pairs

$O(\#inserted \cdot \log n)$

```
3 A .insert({ {5,"K"},  
             {3,"V"}, {1,"G"} })  
3 A .insert(@inB, @inE) -> 1 G 3 A 5 K  
source container 5 K 3 Y 1 G 4 X
```

Insert & Construct Key-Value Pair

$O(\log n)$

```
1 F 3 A .emplace(2,"W") -> {@inserted,true}  
1 F 3 A .emplace_hint(@hint,2,"W") -> {@inserted}  
1 F 3 A .try_emplace(2,"W") -> {@inserted,true}
```

potential performance benefit by hinting at probable insert position

C++17

advantage: does not move from rvalue input parameters if not inserted

Erase Key-Value-Pair(s)

```
1 F 3 A 5 X .erase(2) -> 0  
1 F 3 A 5 X .erase(3) -> 1  
1 F 3 A 5 X .erase(@pos) -> @after_erased  
1 F 3 A 5 X .erase(@b,@e) -> @after_erased
```

$O(\log n)$ $\#erased$
 $O(1)$ amortized
 $O(\log n + \#deleted)$

Erase All

```
1 F 3 A .clear() ->
```

Access / Modify Value

$O(\log n)$

```
map<int,string> m { {1,"F"}, {3,"A"} };  
1 F 3 A [3] -> "A"  
1 F 3 A [3] = "X" -> 1 F 3 X  
Attention: [k] inserts new pair if key k is not present!  
1 F 3 A [2] = "W" -> 1 F 2 W 3 A  
1 F 3 A [2] -> ""
```

Insert or Assign Value

$O(\log n)$ C++17

```
1 F 3 B .insert_or_assign(3,"X") -> {@as,false}  
1 F 3 B .insert_or_assign(5,"R") -> {@ins,true}  
1 F 3 B .insert_or_assign(@hint,3,"W") -> @as  
1 F 3 B .insert_or_assign(@hint,2,"G") -> @ins
```

(inserted yes/no)

Merge Two Maps

$O(n_2 \cdot \log(n_1 + n_2))$ C++17

```
map<int,string> m1 { {1,"F"}, {3,"S"}, {5,"T"} };  
map<int,string> m2 { {2,"A"}, {5,"X"} };  
1 F 3 S 5 T .merge(2 A 5 X)  
1 F 2 A 3 S 5 T
```

Extract Nodes

Allows efficient transfer of key-value pairs. C++17

```
1 F 2 R 3 A .extract(2) -> 2 R  
1 F 2 R 3 A .extract(@pos) -> 2 R
```

$O(\log n)$
 $O(1)$

(Re-)Insert Nodes

members of the return type

```
1 F 3 A .insert(5 N) -> {@position|.inserted|.node}  
1 F 3 A .insert(3 Z) -> {@position|.inserted|.node}  
1 F 3 A .insert() -> {@position|.inserted|.node}  
1 F 3 A .insert(@hint,5 X) -> {@inserted}  
1 F 3 A .insert(@hint,1 G) -> {@blocking}
```

true (empty)
false (empty)

Modify Key

```
map<int,string> m { {1,"F"}, {3,"A"} };  
Direct key modification not allowed!  
auto node = m.extract(3);  
if (node) { // if key existed  
    node.key() = 8;  
    m.insert(move(node));  
}
```

Instead:

- extract
- modify
- re-insert

std::unordered_map

Construct A New Map Object

```
unordered_map<string,int> m0 {}  
unordered_map<string,int> m1 {{ "A",2},  
                               {"Z",4}, {"Y",7}}  
unordered_map<string,int> m2 (begin(m1)+1,  
                               end(m1))  
unordered_map<string,int> deep_copy_of_m1(m1)  
  
C++17 key and mapped types deducible from arguments  
unordered_map<int,double> m3 {{2,3.14},{5,6.0}}; // unordered_map<int,double>
```

Assign New Content To An Existing Map

```
unordered_map<string,int> m1 {{ "X",2}};  
unordered_map<string,int> m2 {{ "A",1}, {"G",4}};  
m1 = m2;  
  
new state of m1  
X2 = A1 G4
```

Lookup Using Keys as Input

$O(1)$ average, $O(n)$ worst case

```
unordered_map<string,int> m {{ "S",3}, {"X",5}, {"F",1}};  
  
S3 F1 X5 .contains("W") → false  
S3 F1 X5 .contains("X") → true  
  
S3 F1 X5 .count("W") → 0  
S3 F1 X5 .count("X") → 1  
  
S3 F1 X5 .find("W") → @end (=no match)  
S3 F1 X5 .find("X") → @match  
  
S3 F1 X5 .equal_range("F") → {@1st_equal, @after}  
S3 F1 X5 .at("F") → 1  
S3 F1 X5 .at("B") → Throws std::out_of_range
```

Query Size

```
F1 A3 .empty() → false  
F1 A3 .size() → 2
```

Erase All

```
F1 A3 .clear()
```

Obtain Iterators

```
F1 A3 .begin() → @first  
F1 A3 .end() → @one_behind_last
```

Insert A Single Key-Value Pair

$O(1)$ avg., $O(n)$ worst

```
F1 A3 .insert({ "W",2}) → {@inserted, true}  
F1 A3 .insert({ "A",9}) → {@blocking, false}  
  
potential performance benefit by hinting at probable insert position  
F1 A3 .insert(@hint, { "W",2}) → @instd/blockng
```

Insert Multiple Key-Value Pairs

$O(\#ins)$ avg., $O(n \cdot \#ins + \#ins)$ worst

```
A3 .insert({ "G",4}, {"K",9}, {"A",7}) → K9 A3 G4  
A3 .insert(@inBegin, @inEnd) → K9 A3 G4  
  
source container G4 K9 A7 X2
```

Construct Key-Value Pair

$O(1)$ avg., $O(n)$ worst

```
F1 A3 .emplace("W",2) → {@inserted, true}  
  
potential performance benefit by hinting at probable insert position  
F1 A3 .emplace_hint(@hint, "W",2) → @inserted  
  
F1 A3 .try_emplace("W",2) → {@inserted, true}  
  
C++17 advantage: does not move from rvalue input parameters if not inserted
```

Insert or Assign Value

$O(1)$ avg., $O(n)$ worst

```
F1 B3 .insert_or_assign("B",5) → {@as, false}  
F1 B3 .insert_or_assign("R",6) → {@ins, true}  
  
potential performance benefit by hinting at probable insert position  
F1 B3 .insert_or_assign(@hint, "B",5) → @as  
F1 B3 .insert_or_assign(@hint, "G",2) → @ins
```

Query & Control Hash Table Properties

hash function: $h(key) \mapsto \text{bucket index}$

```
unordered_map<string,int> um {{ "E",6}, {"X",4}, {"Z",1}, {"A",3}, {"F",2}, {"B",2}};  
  
um .bucket("E") → 4 (key → hash bucket index)  
um .begin(4) → @first_in_bucket  
um .end(4) → @one_behind_last_in_bucket  
um .bucket_size(4) → 2  
um .bucket_count() → 7  
  
um .load_factor() → 4/7 = 0.57  
um .max_load_factor(0.8) (set)  
um .max_load_factor() → 0.8 (get)  
  
(make table large enough to handle min_capacity elements)  
um .reserve(min_capacity) →  
um .rehash(#hash_buckets) →  
  
(can allocate even more buckets if max. load factor demands it)
```

Access / Modify Value

$O(1)$ avg., $O(n)$ worst

```
unordered_map<string,int> m {{ "F",1}, {"A",3}};  
m ["A"] → 3  
m ["A"] = 4 → F1 A4  
  
Attention: [k] inserts new pair if key k is not present!  
m ["W"] = 2 → F1 W2 A3  
m ["W"] → 0  
  
newly created mapped values are value-initialized (e.g. 0 for int)
```

Erase Key-Value-Pair(s)

$O(\#erased)$ avg., $O(n)$ worst case

```
F1 A3 X5 .erase("W") → 0  
F1 A3 X5 .erase("A") → 1  
  
F1 A3 X5 .erase(@pos) → @after  
F1 A3 X5 .erase(@beg, @end) → @after
```

Modify Key

```
unordered_map<string,int> m {{ "F",1}, {"A",3}};  
auto node = m.extract("A");  
if (node) { node.key() = "X";  
            X3  
            m.insert(move(node));  
            F1 X3 }  
  
F1 A3
```

Extract Nodes

to efficiently transfer key-value pairs

C++17

```
F1 R2 A3 .extract("R") → R2  
F1 R2 A3 .extract(@pos) → R2  
  
 $O(1)$  avg.,  $O(n)$  worst
```

(Re-)Insert Nodes

```
F1 A3 .insert(N5) → {@position, inserted, node}  
F1 A3 .insert(A6) → {@position, inserted, node}  
F1 A3 .insert() → {@position, inserted, node}  
F1 A3 .insert(@hint, X5) → @inserted  
F1 A3 .insert(@hint, F6) → @blocking
```

Merge Two Maps

C++17

```
unordered_map<string,int> m1 {{ "F",1}, {"S",3}, {"X",5}};  
unordered_map<string,int> m2 {{ "A",2}, {"X",7}};  
m1 .merge(m2)  
  
 $O(n_2)$  average,  
 $O(n_1 \cdot n_2 + n_2)$  worst case
```