

Программирование на языке C++

Лекция 4

C++

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 - 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 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};
```

Передача структуры как параметр в функцию

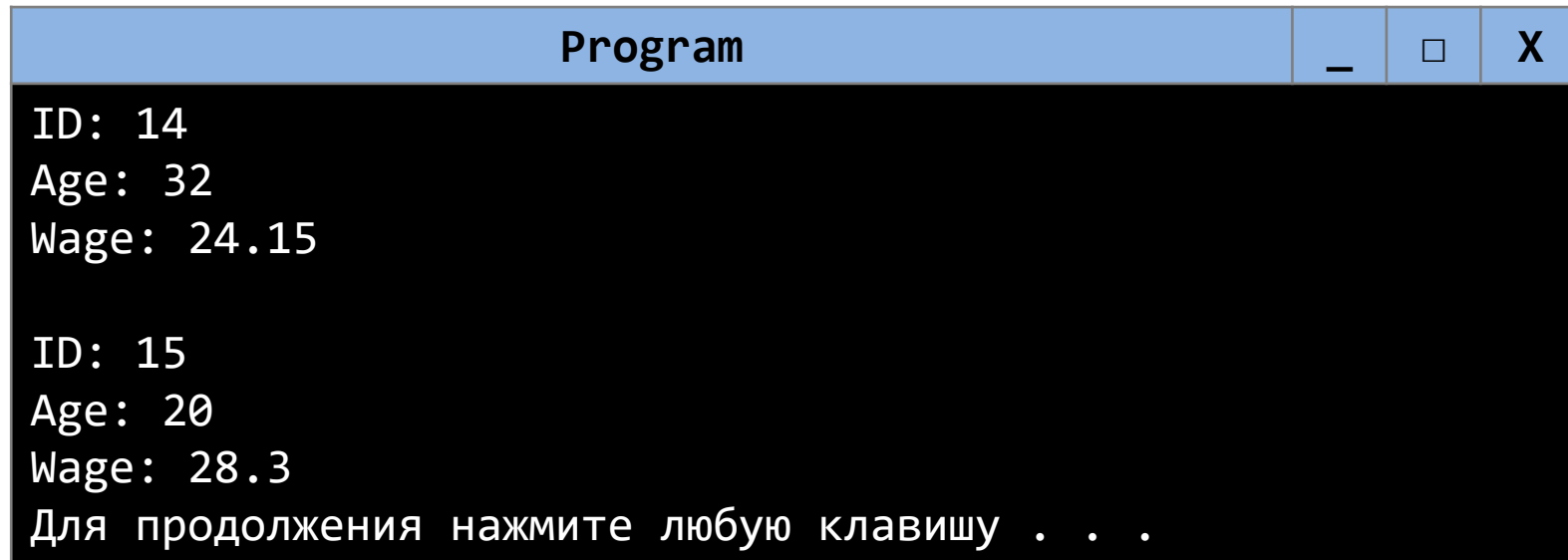
```
struct Employee {  
    short id;  
    int age;  
    double wage;  
};
```

```
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



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

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

The image shows a screenshot of a Windows console window. The title bar is light blue and contains the text "Program" followed by standard window control buttons (minimize, maximize, close). The console area has a black background with white text. 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 . . .".

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;
```

Вложенные структуры

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

```
struct Company {  
    Employee CEO;    // CEO – это структура  
    int numberOfEmployees;  
};
```

```
Company myCompany = {{ 1, 42, 60000.0f }, 5 };  
std::cout << myCompany.CEO.id;
```

Размер структуры и выравнивание I

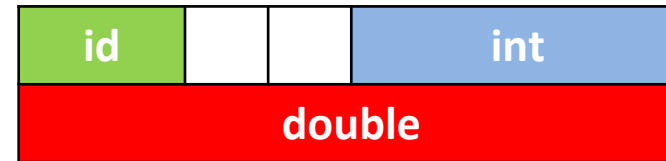
```
struct Employee {  
    short id;    // sizeof(short) == 2  
    int age;     // sizeof(int) == 4  
    double wage; // sizeof(double) == 8  
};  
  
sizeof(Employee); // 16 != ( 2 + 4 + 8 )
```

Размер структуры и выравнивание II

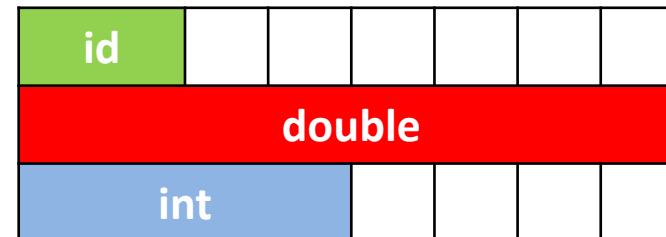
```
struct Employee {  
    short id;      // sizeof(short) == 2  
    double wage;   // sizeof(double) == 8  
    int age;       // sizeof(int) == 4  
};  
  
sizeof(Employee); // 24 != ( 2 + 4 + 8 )
```

Размер структуры и выравнивание II

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



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



Объединения

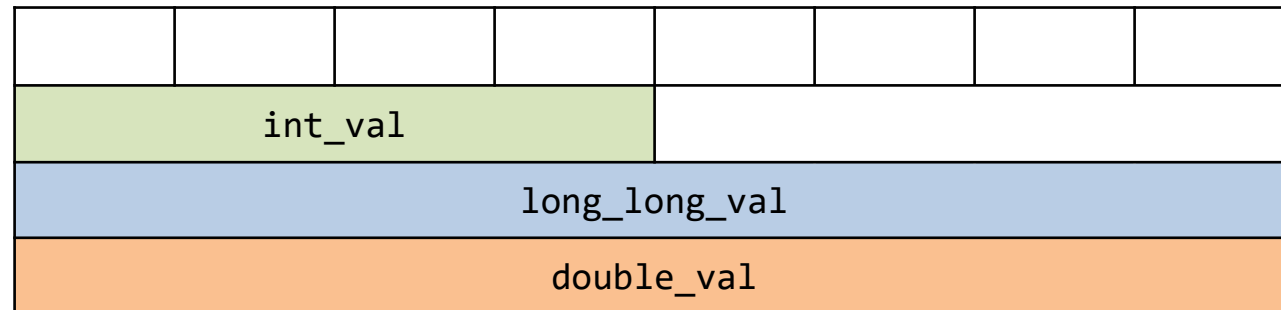
Что такое объединение?

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
}
```

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:

```
Colors 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:

```
Colors dor = YELLOW;
```

```
c = dor;
```

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

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

Операции II

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

Преобразования в перечисление

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

```
Colors 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);
```


Enum class

```
#include <iostream>

enum class Color{
    YELLOW,
    BLACK,
    PINK
};

enum class Feelings{
    SAD,
    ANGRY,
    BLACK
};

int main(){
    std::cout << Color::BLACK;    // Нормально
    std::cout << Feelings::BLACK; // Нормально
    std::cout << BLACK;           // Ошибка
}
```

Тот же Enum, но перечислители объявляются не в той же области видимости, что и Enum, а заворачиваются в пространство имён с именем перечисления.

Ассоциативные контейнеры

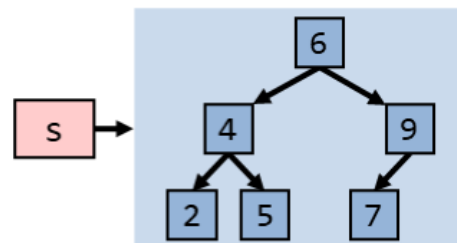
set<Key>

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)

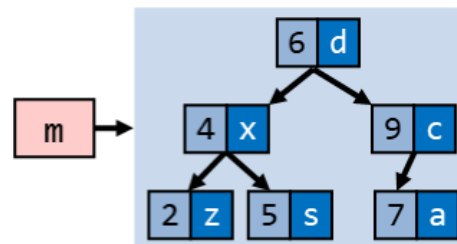
map<Key, Value>

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)

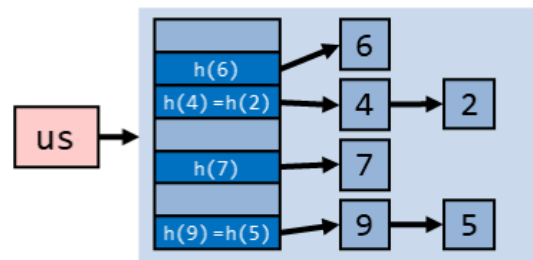
unordered_set<Key>

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

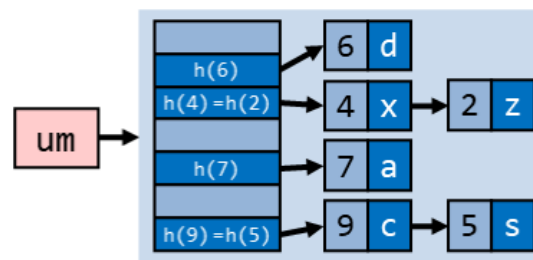
unordered_map<Key, Value>

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 } can be > 1 only for std::multiset  
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}
```

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

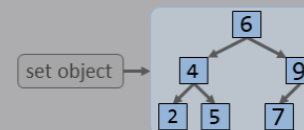
$O(\log n)$

```
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 } #erased  
1 3 5 7 .erase(@pos) → @after_erased  
1 3 5 7 .erase(@beg, @end) → @after_erased  
 $O(\log n)$   
 $O(1)$  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)

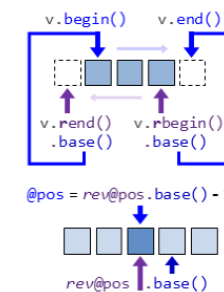


Obtain Iterators

```
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
```

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
```

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
```

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)  
US .begin(4) → @first_in_bucket  
US .end(4) → @one_behind_last_in_bucket  
US .bucket_size(4) → 2  
US .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
```

Insert Nodes

C++17

members of the return type

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

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);  
node.value() = 8;  
s.insert(move(node));
```

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}
```

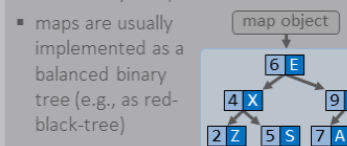
potential performance benefit by hinting at probable insert position

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
```

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
```

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] -> ""  
1 F 2 3 A
```

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
```

potential performance benefit by hinting at probable insert position

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 5 X
```

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
```

empty node
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 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}) → @insert/blocking
```

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

C++17

```
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");  
A3 F1  
if (node) { node.key() = "X";  
X3 F1  
m.insert(move(node));  
F1 X3 }
```

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 N5 true (empty)  
  
F1 A3 .insert(A6) → {@position, inserted, node}  
F1 A3 A6 false Z4  
  
F1 A3 .insert() → {@position, inserted, node}  
F1 A3 false (empty)  
  
F1 A3 .insert(@hint, X5) → @inserted  
F1 A3 X5  
  
F1 A3 .insert(@hint, F6) → @blocking  
F1 A3
```

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}};  
  
F1 S3 X5 .merge(A2 X7)  
F1 A2 S3 X5 X7  
  
 $O(n_2)$  average,  
 $O(n_1 \cdot n_2 + n_2)$  worst case
```