

Лекція 28.

Алгоритми для роботи з множинами та приклади їх використання. Smart Pointers.



План на сьогодні

1

Алгоритм includes

2

Алгоритм set_union

3

Алгоритм set_intersection

4

Алгоритм set_difference

5

Алгоритм set_symmetric_difference

6

Smart Pointers



Алгоритм includes

includes

- ❑ У мові C++ `includes()` — це стандартна функція, яка перевіряє, чи всі елементи одного відсортованого діапазону містяться в іншому.
- ❑ Працює лише з **відсортованими** діапазонами, інакше результат буде некоректним.
- ❑ Потребує заголовку `#include <algorithm>`.
- ❑ Може використовувати **власну функцію порівняння (предикат)**.

Синтаксис:

- ❑ `includes(first1, last1, first2, last2);`
- ❑ `includes(first1, last1, first2, last2, comp);`

Параметри:

- ❑ `first1, last1`: Ітератори на перший (основний) відсортований діапазон.
- ❑ `first2, last2`: Ітератори на другий відсортований діапазон, який перевіряється як підмножина.
- ❑ `comp`: (необов'язково) Функція порівняння, яка визначає порядок.

Повертає:

- ❑ Якщо всі елементи другого діапазону знайдено у першому — повертає `true`.
- ❑ Якщо хоча б один елемент не знайдено — повертає `false`.

Реалізація includes

```
template<class InputIt1, class InputIt2>
bool includes(InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2)
{
    for (; first2 != last2; ++first1)
    {
        if (first1 == last1 || *first2 < *first1)
            return false;
        if (!(*first1 < *first2))
            ++first2;
    }
    return true;
}
```

Реалізація includes з предикатом

```
template<class InputIt1, class InputIt2, class Compare>
bool includes(InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2, Compare comp)
{
    for (; first2 != last2; ++first1)
    {
        if (first1 == last1 || comp(*first2, *first1))
            return false;
        if (!comp(*first1, *first2))
            ++first2;
    }
    return true;
}
```

Приклад 1

```
int main()
{
    // lottery numbers
    vector<int> lottery = { 1, 4, 6, 3, 2, 54, 32 };

    // Numbers in user's card
    vector<int> user = { 1, 2, 4, 6 };

    // sorting initial containers
    sort(lottery.begin(), lottery.end());
    sort(user.begin(), user.end());

    // using include() check if all elements
    // of user are present as lottery numbers
    if (includes(lottery.begin(), lottery.end(),
                user.begin(), user.end()))
        cout << "User has won lottery ( all numbers are "
                "lottery numbers )";
    else
        cout << "User has not won the lottery";
}
```

User has won lottery (all numbers are lottery numbers)

Приклад 2 (з предикатом)

```
struct Person {  
    std::string name;  
    int age;  
  
    Person(std::string n, int a) : name(std::move(n)), age(a) {}  
};
```

// Comparator: compare persons by age only

```
struct CompareByAge {  
    bool operator()(const Person& a, const Person& b) const {  
        return a.age < b.age;  
    }  
};
```

// Overload output stream operator for displaying Person

```
std::ostream& operator<<(std::ostream& os, const Person& p) {  
    return os << p.name << " (" << p.age << ")";  
}
```

```
int main() {
```

```
    // Main set of people (sorted by age using custom comparator)  
    std::set<Person, CompareByAge> groupA = {  
        {"Alice", 30}, {"Bob", 25}, {"Charlie", 35}, {"Dana", 28}  
    };
```

// Subset of people to check for inclusion

```
    std::set<Person, CompareByAge> groupB = {  
        {"Unknown", 25}, {"Someone", 30}  
    };
```

// Use std::includes with custom comparator to check if groupB is a subset of groupA

```
    if (std::includes(groupA.begin(), groupA.end(),  
        groupB.begin(), groupB.end(), CompareByAge())) {  
        std::cout << "groupB is a subset of groupA by age\n";  
    }  
    else {  
        std::cout << "groupB is not a subset of groupA\n";  
    }  
}
```

groupB is a subset of groupA by age

Алгоритм set_union

set_union

- ❑ У мові C++ `set_union()` — це стандартна функція з бібліотеки `<algorithm>`, яка об'єднує два **відсортовані** діапазони в один.
- ❑ Результат міститиме всі елементи з обох діапазонів без дублювання однакових значень.
- ❑ Для правильного результату обидва діапазони повинні бути **відсортовані за однаковим критерієм**.

Синтаксис:

- ❑ `set_union(first1, last1, first2, last2, result);`
- ❑ `set_union(first1, last1, first2, last2, result, comp);`

Параметри:

- ❑ `first1, last1`: Ітератори на перший відсортований діапазон.
- ❑ `first2, last2`: Ітератори на другий відсортований діапазон.
- ❑ `result`: Ітератор, куди буде записано результат (початок області призначення).
- ❑ `comp`: (необов'язково) Користувацька функція порівняння.

Повертає:

- ❑ Ітератор на кінець записаного результату у вихідному діапазоні.

Реалізація set_union

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt set_union(InputIt1 first1, InputIt1 last1,
                  InputIt2 first2, InputIt2 last2, OutputIt d_first)
{
    for (; first1 != last1; ++d_first)
    {
        if (first2 == last2)
            return std::copy(first1, last1, d_first);

        if (*first2 < *first1)
            *d_first = *first2++;
        else
        {
            *d_first = *first1;
            if (!(*first1 < *first2))
                ++first2;
            ++first1;
        }
    }
    return std::copy(first2, last2, d_first);
}
```

Реалізація set_union з предикатом

```
template<class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt set_union(InputIt1 first1, InputIt1 last1,
                  InputIt2 first2, InputIt2 last2, OutputIt d_first, Compare comp)
{
    for (; first1 != last1; ++d_first)
    {
        if (first2 == last2)
            // Finished range 2, include the rest of range 1:
            return std::copy(first1, last1, d_first);

        if (comp(*first2, *first1))
            *d_first = *first2++;
        else
        {
            *d_first = *first1;
            if (!comp(*first1, *first2)) // Equivalent => don't need to include *first2.
                ++first2;
            ++first1;
        }
    }
    // Finished range 1, include the rest of range 2:
    return std::copy(first2, last2, d_first);
}
```

Приклад 3

Combined permissions:

- delete
- execute
- read
- share
- write

```
int main() {  
    // Permissions from Role A  
    std::vector<std::string> roleA = { "read", "write", "execute" };  
  
    // Permissions from Role B  
    std::vector<std::string> roleB = { "write", "delete", "share" };  
  
    // Sort both vectors (required for set_union)  
    std::sort(roleA.begin(), roleA.end());  
    std::sort(roleB.begin(), roleB.end());  
  
    // Vector to store the union result  
    std::vector<std::string> allPermissions;  
  
    // Perform set_union  
    std::set_union(  
        roleA.begin(), roleA.end(),  
        roleB.begin(), roleB.end(),  
        std::back_inserter(allPermissions)  
    );  
  
    // Print the result  
    std::cout << "Combined permissions:\n";  
    for (const auto& perm : allPermissions) {  
        std::cout << "- " << perm << "\n";  
    }  
}
```

Приклад 4. (з предикатом)

```
int main() {
    // Group A
    std::vector<Person> groupA = {
        {"Alice", 30},
        {"Bob", 25},
        {"Charlie", 35},
        {"Bob", 40} // Same name as another Bob but different age
    };

    // Group B
    std::vector<Person> groupB = {
        {"Bob", 25},    // Duplicate (exact match with groupA)
        {"Bob", 40},    // Duplicate (exact match with groupA)
        {"Dana", 28},
        {"Alice", 22}   // Same name, different age → considered different
    };

    // Sort both vectors using combined comparator
    std::sort(groupA.begin(), groupA.end(), CompareByNameAndAge());
    std::sort(groupB.begin(), groupB.end(), CompareByNameAndAge());

    // Result vector
    std::vector<Person> mergedGroup;

    std::set_union(
        groupA.begin(), groupA.end(),
        groupB.begin(), groupB.end(),
        std::back_inserter(mergedGroup),
        CompareByNameAndAge()
    );

    std::cout << "Merged group (unique by name + age):\n";
    for (const auto& person : mergedGroup) {
        std::cout << "- " << person << "\n";
    }
}
```

```
struct Person {
    std::string name;
    int age;

    Person(std::string n, int a) : name(std::move(n)), age(a) {}
};

// Comparator: compare by both name and age
struct CompareByNameAndAge {
    bool operator()(const Person& a, const Person& b) const {
        if (a.name != b.name)
            return a.name < b.name;
        return a.age < b.age;
    }
};

std::ostream& operator<<(std::ostream& os, const Person& p) {
    return os << p.name << " (" << p.age << ")";
}
```

```
Merged group (unique by name + age):
- Alice (22)
- Alice (30)
- Bob (25)
- Bob (40)
- Charlie (35)
- Dana (28)
```

Алгоритм set_intersection

set_intersection

- ❑ У мові C++ `set_intersection()` — це стандартна функція з бібліотеки `<algorithm>`, яка знаходить спільні елементи між двома **відсортованими** діапазонами.
- ❑ Результат містить лише ті елементи, які є в **обох** діапазонах.
- ❑ Обидва діапазони мають бути **відсортовані за однаковим порядком**.

Синтаксис:

- ❑ `set_intersection(first1, last1, first2, last2, result);`
- ❑ `set_intersection(first1, last1, first2, last2, result, comp);`

Параметри:

- ❑ `first1, last1`: Ітератори на перший відсортований діапазон.
- ❑ `first2, last2`: Ітератори на другий відсортований діапазон.
- ❑ `result`: Ітератор, куди буде записано результат (початок області призначення).
- ❑ `comp`: (необов'язково) Користувацька функція порівняння..

Повертає:

- ❑ Ітератор на кінець записаного результату у вихідному діапазоні.

Реалізація set_intersection

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt set_intersection(InputIt1 first1, InputIt1 last1,
                          InputIt2 first2, InputIt2 last2, OutputIt d_first)
{
    while (first1 != last1 && first2 != last2)
    {
        if (*first1 < *first2)
            ++first1;
        else
        {
            if (!(*first2 < *first1))
                *d_first++ = *first1++; // *first1 and *first2 are equivalent.
            ++first2;
        }
    }
    return d_first;
}
```

Реалізація set_intersection з предикатом

```
template<class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt set_intersection(InputIt1 first1, InputIt1 last1,
                          InputIt2 first2, InputIt2 last2, OutputIt d_first, Compare comp)
{
    while (first1 != last1 && first2 != last2)
    {
        if (comp(*first1, *first2))
            ++first1;
        else
        {
            if (!comp(*first2, *first1))
                *d_first++ = *first1++; // *first1 and *first2 are equivalent.
            ++first2;
        }
    }
    return d_first;
}
```

Приклад 5

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

using namespace std;

int main() {
    vector<int> numbersA = {1, 2, 4, 5, 6};
    vector<int> numbersB = {2, 5, 7};
    vector<int> common;

    set_intersection(numbersA.begin(), numbersA.end(),
                     numbersB.begin(), numbersB.end(),
                     back_inserter(common));

    for (size_t i = 0; i < common.size(); ++i)
        cout << common[i] << " "; // Виведе: 2 5

    return 0;
}
```

2 5

Приклад 6 (з предикатом)

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

using namespace std;

bool compareDescending(int a, int b) {
    return a > b;
}

int main() {
    vector<int> numbersA = {6, 5, 4, 2, 1};
    vector<int> numbersB = {7, 5, 2};
    vector<int> common;

    set_intersection(numbersA.begin(), numbersA.end(),
                     numbersB.begin(), numbersB.end(),
                     back_inserter(common), compareDescending);

    for (size_t i = 0; i < common.size(); ++i)
        cout << common[i] << " "; // Виведе: 5 2

    return 0;
}
```

5 2

Приклад 7 (з предикатом)

```
struct Person {
    std::string name;
    int age;

    Person(std::string n, int a) : name(std::move(n)), age(a) {}
};

// Comparator for full ordering: by name, then age
struct CompareByNameAndAge {
    bool operator()(const Person& a, const Person& b) const {
        if (a.name != b.name)
            return a.name < b.name;
        return a.age < b.age;
    }
};

std::ostream& operator<<(std::ostream& os, const Person& p) {
    return os << p.name << " (" << p.age << ")";
}
```

People present in both sets (same name + age):

- Bob (40)
- Charlie (35)

```
int main() {
    std::set<Person, CompareByNameAndAge> usersA = {
        {"Alice", 30},
        {"Bob", 25},
        {"Bob", 40},
        {"Charlie", 35}
    };

    std::set<Person, CompareByNameAndAge> usersB = {
        {"Bob", 40},
        {"Bob", 50},
        {"Charlie", 35},
        {"Dana", 28}
    };

    // Set to store intersection (identical people)
    std::set<Person, CompareByNameAndAge> intersectionResult;
    // Perform set_intersection directly
    std::set_intersection(
        usersA.begin(), usersA.end(),
        usersB.begin(), usersB.end(),
        std::inserter(intersectionResult, intersectionResult.begin()),
        CompareByNameAndAge());

    std::cout << "People present in both sets (same name + age):\n";
    for (const auto& p : intersectionResult) {
        std::cout << "- " << p << "\n";
    }
}
```

Алгоритм set_difference

set_difference

- ❑ У мові C++ `set_difference()` — це стандартна функція з бібліотеки `<algorithm>`, яка обчислює різницю між двома **відсортованими** діапазонами.
- ❑ Вона повертає елементи, які присутні у **першому** діапазоні, але **відсутні** у другому.
- ❑ Обидва діапазони повинні бути **відсортовані** відповідно до одного й того ж критерію.

Синтаксис:

- ❑ `set_difference(first1, last1, first2, last2, result);`
- ❑ `set_difference(first1, last1, first2, last2, result, comp);`

Параметри:

- ❑ `first1, last1`: Ітератори на перший відсортований діапазон.
`first2, last2`: Ітератори на другий відсортований діапазон.
- ❑ `result`: Ітератор, куди буде записано результат (початок області призначення).
- ❑ `comp`: (необов'язково) Користувачька функція порівняння..

Повертає:

- ❑ Ітератор на кінець записаного результату у вихідному діапазоні.

Реалізація set_difference

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt set_difference(InputIt1 first1, InputIt1 last1,
                       InputIt2 first2, InputIt2 last2, OutputIt d_first)
{
    while (first1 != last1)
    {
        if (first2 == last2)
            return std::copy(first1, last1, d_first);

        if (*first1 < *first2)
            *d_first++ = *first1++;
        else
        {
            if (! (*first2 < *first1))
                ++first1;
            ++first2;
        }
    }
    return d_first;
}
```


Реалізація set_difference з предикатом

```
template<class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt set_difference(InputIt1 first1, InputIt1 last1,
                       InputIt2 first2, InputIt2 last2, OutputIt d_first, Compare comp)
{
    while (first1 != last1)
    {
        if (first2 == last2)
            return std::copy(first1, last1, d_first);

        if (comp(*first1, *first2))
            *d_first++ = *first1++;
        else
        {
            if (!comp(*first2, *first1))
                ++first1;
            ++first2;
        }
    }
    return d_first;
}
```

Приклад 8

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

using namespace std;

int main() {
    vector<int> numbersA = {1, 2, 3, 4, 5};
    vector<int> numbersB = {3, 4, 6};
    vector<int> result;

    set_difference(numbersA.begin(), numbersA.end(),
                  numbersB.begin(), numbersB.end(),
                  back_inserter(result));

    for (size_t i = 0; i < result.size(); ++i)
        cout << result[i] << " "; // Виведе: 1 2 5

    return 0;
}
```



1 2 5

Приклад 9 (з предикатом)

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

using namespace std;

bool compareDescending(int a, int b) {
    return a > b;
}

int main() {
    vector<int> numbersA = {6, 5, 4, 2, 1};
    vector<int> numbersB = {5, 1};
    vector<int> result;

    set_difference(numbersA.begin(), numbersA.end(),
                  numbersB.begin(), numbersB.end(),
                  back_inserter(result), compareDescending);

    for (size_t i = 0; i < result.size(); ++i)
        cout << result[i] << " "; // Виведе: 6 4 2

    return 0;
}
```



6 4 2

Приклад 10

```
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& v)
{
    os << '{';
    for (auto it = v.begin(); it != v.end(); ++it) {
        os << *it;
        if (std::next(it) != v.end()) {
            os << ", ";
        }
    }
    return os << '}';
}

struct Order
{
    int order_id{};
    std::string customer_name;
    double amount{};

    // Comparison by order_id for algorithms like set_difference
    bool operator<(const Order& other) const {
        return order_id < other.order_id;
    }

    // Output formatting
    friend std::ostream& operator<<(std::ostream& os, const Order& ord)
    {
        return os << "Order{id=" << ord.order_id
            << ", name=" << ord.customer_name
            << ", amount=" << ord.amount << "}";
    }
};
```

```
int main()
{
    const std::vector<int> v1{ 1, 2, 5, 5, 5, 9 };
    const std::vector<int> v2{ 2, 5, 7 };
    std::vector<int> diff;
    std::set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
        std::inserter(diff, diff.begin()));
    std::cout << v1 << " / " << v2 << " == " << diff << "\n\n";

    std::vector<Order> old_orders{
        {1, "Alice", 120.0},
        {2, "Bob", 99.5},
        {5, "Charlie", 30.0},
        {9, "Diana", 250.0}
    };
    std::vector<Order> new_orders{
        {2, "Bob", 99.5},
        {5, "Charlie", 30.0},
        {7, "Eva", 88.8}
    };
    std::vector<Order> cut_orders;
    std::set_difference(old_orders.begin(), old_orders.end(),
        new_orders.begin(), new_orders.end(),
        std::back_inserter(cut_orders));
    std::cout << "old orders: " << old_orders << '\n'
        << "new orders: " << new_orders << '\n'
        << "cut orders: " << cut_orders << '\n';
}
```

```
{1, 2, 5, 5, 5, 9} / {2, 5, 7} == {1, 5, 5, 9}
```

```
old orders: {Order{id=1, name=Alice, amount=120}, Order{id=2, name=Bob, amount=99.5}, Order{id=5, name=Charlie, amount=30}, Order{id=9, name=Diana, amount=250}}
new orders: {Order{id=2, name=Bob, amount=99.5}, Order{id=5, name=Charlie, amount=30}, Order{id=7, name=Eva, amount=88.8}}
cut orders: {Order{id=1, name=Alice, amount=120}, Order{id=9, name=Diana, amount=250}}
```

Алгоритм set_symmetric_difference



set_symmetric_difference

- ❑ У мові C++ `set_symmetric_difference()` — це функція з бібліотеки `<algorithm>`, яка обчислює **симетричну різницю** між двома **відсортованими** діапазонами.
- ❑ Симетрична різниця — це елементи, які належать **лише одному** з діапазонів, але **не обом**.
- ❑ Для правильного результату обидва діапазони повинні бути **відсортовані за однаковим порядком**.

Синтаксис:

- ❑ `set_symmetric_difference(first1, last1, first2, last2, result);`
- ❑ `set_symmetric_difference(first1, last1, first2, last2, result, comp);`

Параметри:

- ❑ `first1, last1`: Ітератори на перший відсортований діапазон.
- ❑ `first2, last2`: Ітератори на другий відсортований діапазон.
- ❑ `result`: Ітератор на початок діапазону, куди буде записано результат.
- ❑ `comp`: (необов'язково) Функція порівняння (предикат).

Повертає:

- ❑ Ітератор на кінець записаного результату.

Реалізація set_symmetric_difference

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt set_symmetric_difference(InputIt1 first1, InputIt1 last1,
                                InputIt2 first2, InputIt2 last2, OutputIt d_first)
{
    while (first1 != last1)
    {
        if (first2 == last2)
            return std::copy(first1, last1, d_first);

        if (*first1 < *first2)
            *d_first++ = *first1++;
        else
        {
            if (*first2 < *first1)
                *d_first++ = *first2;
            else
            {
                ++first1;
                ++first2;
            }
        }
    }
    return std::copy(first2, last2, d_first);
}
```

Реалізація set_symmetric_difference з предикатом

```
template<class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt set_symmetric_difference(InputIt1 first1, InputIt1 last1,
                                  InputIt2 first2, InputIt2 last2,
                                  OutputIt d_first, Compare comp)
{
    while (first1 != last1)
    {
        if (first2 == last2)
            return std::copy(first1, last1, d_first);

        if (comp(*first1, *first2))
            *d_first++ = *first1++;
        else
        {
            if (comp(*first2, *first1))
                *d_first++ = *first2;
            else
            {
                ++first1;
                ++first2;
            }
        }
    }
    return std::copy(first2, last2, d_first);
}
```


Приклад 11

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

using namespace std;

int main() {
    vector<int> numbersA = {1, 2, 3, 4};
    vector<int> numbersB = {3, 4, 5, 6};
    vector<int> result;

    set_symmetric_difference(numbersA.begin(), numbersA.end(),
                             numbersB.begin(), numbersB.end(),
                             back_inserter(result));

    for (size_t i = 0; i < result.size(); ++i)
        cout << result[i] << " "; // Виведе: 1 2 5 6

    return 0;
}
```

1 2 5 6

Приклад 12 (з предикатом)

```
struct Order {
    int order_id{};
    std::string customer_name;
    double amount{};

    // Compare by order_id
    bool operator<(const Order& other) const {
        return order_id < other.order_id;
    }

    // Output formatting
    friend std::ostream& operator<<(std::ostream& os, const Order& ord) {
        return os << "Order{id=" << ord.order_id
            << ", name=" << ord.customer_name
            << ", amount=" << ord.amount << "}";
    }
};
```

```
Orders that were added or removed:
Order{id=1, name=Alice, amount=120}
Order{id=7, name=Eva, amount=88.8}
Order{id=9, name=Diana, amount=250}
Order{id=10, name=Frank, amount=60}
```

```
int main() {
    std::set<Order> old_orders{
        {1, "Alice", 120.0},
        {2, "Bob", 99.5},
        {5, "Charlie", 30.0},
        {9, "Diana", 250.0}
    };
    std::set<Order> new_orders{
        {2, "Bob", 99.5},
        {5, "Charlie", 30.0},
        {7, "Eva", 88.8},
        {10, "Frank", 60.0}
    };
    std::set<Order> changed_orders;
    std::set_symmetric_difference(
        old_orders.begin(), old_orders.end(),
        new_orders.begin(), new_orders.end(),
        std::inserter(changed_orders, changed_orders.begin())
    );
    std::cout << "Orders that were added or removed:\n";
    for (const auto& o : changed_orders) {
        std::cout << "  " << o << '\n';
    }
}
```

Приклади тестів

```
void F1()
{
    unsigned u[] = { 1,20,40,30,20 };
    set<unsigned> a(u, u + 5);
    set<unsigned> b = a;
    replace_copy(a.begin(), a.end(), inserter(b,b.begin()), 20, 1);
    cout << count(b.begin(), b.end(), *a.begin());
}

void F2()
{
    unsigned u[] = { 2,40,3,2,40 };
    map<unsigned, unsigned> a;
    unsigned i = 5;
    while (--i){
        a[u[i]] = i;
    }
    unsigned s = 0;
    for (map<unsigned, unsigned>::iterator f = a.find(2); f != a.end(); f++)
        s += f->second;
    cout << s;
}
```

Приклади тестів

```
void F3()  
{  
    unsigned u[] = { 1,20,40,30,20 };  
    vector<unsigned> a(u, u + 5);  
    vector<unsigned> b = a;  
    replace_copy(a.begin(), a.end(), b.begin(), 20, 1);  
    cout << count(b.begin(), b.end(), *a.begin());  
}
```

```
void F4()  
{  
    const unsigned MSize = 10;  
    unsigned u[] = { 1,20,20,30,50 };  
    list<unsigned> a(u, u + 5);  
    vector<unsigned> b(MSize);  
    unique_copy(a.begin(), a.end(), b.begin());  
    unsigned s = 0;  
    vector<unsigned>::iterator f = b.begin();  
    while (f != b.end()) s += *f++;  
    cout << s;  
}
```

Smart Pointers

Smart Pointer

- ❑ **Розумний вказівник (Smart pointer)** — це клас-обгортка, який поводить себе як звичайний вказівник, але додатково автоматично керує життєвим циклом об'єкта, на який він вказує. Головна мета — запобігти витокам пам'яті та автоматизувати звільнення ресурсів.
- ❑ Є частиною заголовочного файлу `<memory>`.
- ❑ Розумні вказівники реалізовані як шаблони, тому ми можемо створювати розумні вказівники на будь-який тип пам'яті.

Бібліотеки C++ надають реалізацію таких типів розумних вказівників:

- ❑ `auto_ptr` - `deprecated` **передає право власності** під час копіювання, це може призвести до **неочікуваної поведінки**, тому з C++11 його **замінили** на `unique_ptr`;
- ❑ `unique_ptr` - гарантує, що об'єктом володіє **лише один власник**;
- ❑ `shared_ptr` - дозволяє **декільком вказівникам** спільно володіти одним об'єктом у пам'яті;
- ❑ `weak_ptr` - це допоміжний розумний вказівник, який **спостерігає** за об'єктом, яким володіє `shared_ptr`, але **не впливає** на його лічильник посилань.

Проблеми звичайних вказівників (T*)

При використанні "сирих" вказівників (raw pointers) T* потрібно завжди пам'ятати про:

- ❑ створення об'єкта через new,
- ❑ видалення через delete,
- ❑ уникнення подвійного delete,
- ❑ уникнення забутих delete (витік пам'яті (memory leak)),
- ❑ правильне звільнення пам'яті при генерації винятків.

Розумні вказівники вирішують ці проблеми за рахунок принципу **RAII (Resource Acquisition Is Initialization)** - ресурси виділяються в момент створення об'єкта і звільняються в його деструкторі.

```
#include <iostream>
#include <stdexcept>
```

```
void bad_example() {
    int* data = new int[100]; // Виділяємо пам'ять

    // Якщо тут виникне виняток, delete[] не буде викликаний
    throw std::runtime_error("Something went wrong!");

    delete[] data; // ← не буде виконано!
}
```

```
void good_example() {
    std::unique_ptr<int[]> data(new int[100]);
    // RAII – пам'ять буде звільнено автоматично

    // Викидаємо виняток
    throw std::runtime_error("Still safe!");
    // Немає delete – не треба!
}
```

Приклад реалізації розумного вказівника

```
template<typename T>
class ScopedPtr {
    T* ptr;
public:
    explicit ScopedPtr(T* p = nullptr) : ptr(p) {}
    // Забороняємо копіювання (як у std::unique_ptr)
    ScopedPtr(const ScopedPtr&) = delete;
    ScopedPtr& operator=(const ScopedPtr&) = delete;
    // Дозволяємо переміщення
    ScopedPtr(ScopedPtr&& other) noexcept : ptr(other.ptr) {
        other.ptr = nullptr;
    }
    ScopedPtr& operator=(ScopedPtr&& other) noexcept {
        if (this != &other) {
            delete ptr;
            ptr = other.ptr;
            other.ptr = nullptr;
        }
        return *this;
    }
    ~ScopedPtr() {
        delete ptr;
    }
    T* operator->() const { return ptr; }
    T& operator*() const { return *ptr; }
    T* get() const { return ptr; }
    T* release() {
        T* temp = ptr;
        ptr = nullptr;
        return temp;
    }
};
```

```
struct MyResource {
    MyResource() { std::cout << "Acquired\n"; }
    ~MyResource() { std::cout << "Released\n"; }
    void doWork() { std::cout << "Working...\n"; }
};

void safe_function() {
    ScopedPtr<MyResource> res(new MyResource);
    res->doWork();

    // Навіть якщо тут виняток – пам'ять буде звільнено
    throw std::runtime_error("Something went wrong!");
}

int main() {
    try {
        safe_function();
    }
    catch (const std::exception& e) {
        std::cout << "Exception: " << e.what() << "\n";
    }
}
```


unique_ptr

- ❑ `unique_ptr` — це розумний вказівник у C++, який гарантує, що об'єктом володіє **лише один власник**.
- ❑ Об'єкт автоматично звільняється, коли `unique_ptr` виходить за межі області видимості.
- ❑ Неможливо копіювати, але можна передавати право власності через `std::move()`
- ❑ `#include <memory>`

Синтаксис:

- ❑ `unique_ptr<T> ptr = make_unique<T>(args...);`

Примітки:

- ❑ Заборонене копіювання: `unique_ptr<T> p2 = p1; // ❌`
Дозволене переміщення: `unique_ptr<T> p2 = move(p1); // ✅`
- ❑ Безпечне створення: `make_unique<T>()`

Приклад unique_ptr

```
#include <iostream>
#include <memory>    // для unique_ptr

using namespace std;

int main() {
    // Створення об'єкта через make_unique
    unique_ptr<int> ptr1 = make_unique<int>(42);
    cout << "Значення ptr1: " << *ptr1 << endl;

    // unique_ptr не можна копіювати:
    // unique_ptr<int> ptr2 = ptr1;    // Помилка компіляції

    // Але можна передати право власності:
    unique_ptr<int> ptr2 = move(ptr1);

    if (!ptr1)
        cout << "ptr1 більше не вказує на об'єкт" << endl;

    cout << "Значення ptr2: " << *ptr2 << endl;

    return 0;
}
```

Значення ptr1: 42
ptr1 більше не вказує на об'єкт
Значення ptr2: 42

Приклад unique_ptr

```
struct Resource {  
    Resource() { std::cout << "Created\n"; }  
    ~Resource() { std::cout << "Destroyed\n"; }  
    void say() { std::cout << "Using resource\n"; }  
};  
  
void use(std::unique_ptr<Resource> r) {  
    r->say();  
}  
  
int main() {  
    auto r = std::make_unique<Resource>();  
    cout << "Before function\n";  
    use(std::move(r)); // Після цього r – порожній (nullptr)  
    cout << "After function\n";  
}
```

```
Created  
Before function  
Using resource  
Destroyed  
After function
```

Чому make_unique а не new?

Ознака	<code>make_unique<T>()</code>	<code>new + unique_ptr</code>
Захист від витоків	✅ Так	❌ Ні
Лаконічність	✅ Так	❌ Ні
Рекомендоване використання	✅ Так (завжди)	⚠️ Уникати, якщо можливо
Підтримка власних deleter'ів	✅ Так (через <code>unique_ptr</code>)	✅ Так

```
struct A {  
    A(int) { throw std::runtime_error("Fail"); }  
};  
  
void unsafe() {  
    // Memory leak! new A(5) is never deleted  
    auto p = std::unique_ptr<A>(new A(5)); // 💣 exception here  
}  
  
void safe() {  
    // No leak – make_unique handles it correctly  
    auto p = std::make_unique<A>(5); // 💣 safe  
}
```

shared_ptr

- ❑ `shared_ptr` — це розумний вказівник у C++, який дозволяє **декільком вказівникам** спільно володіти одним об'єктом у пам'яті.
- ❑ Об'єкт автоматично звільняється, коли останній `shared_ptr`, що на нього посилається, буде знищено або перенаправлений.
- ❑ `#include <memory>`

Синтаксис:

- ❑ `shared_ptr<T> ptr = make_shared<T>(args...);`

Примітки:

- ❑ Дозволяє копіювання: `shared_ptr<T> p2 = p1; // ✓`
- ❑ Веде облік посилань: `use_count()` повертає кількість `shared_ptr`, що вказують на об'єкт.
- ❑ Безпечне створення: `make_shared<T>()` створює об'єкт та вказівник одразу.

Приклад shared_ptr

```
int main() {  
    // Створення спільного розумного вказівника  
    shared_ptr<int> ptr1 = make_shared<int>(100);  
    cout << "Value ptr1: " << *ptr1 << endl;  
    cout << "Pointers counter ptr1: " << ptr1.use_count() << endl;  
  
    {  
        // Створення другого shared_ptr, який ділить право власності  
        shared_ptr<int> ptr2 = ptr1;  
        cout << "Value ptr2: " << *ptr2 << endl;  
        cout << "Pointers counter ptr1: " << ptr1.use_count() << endl;  
    }  
    cout << "After ptr2 deletion\n";  
    // ptr2 виходить з області видимості, але пам'ять НЕ звільняється  
    // поки ptr1 ще існує  
    cout << "Pointers counter ptr1: " << ptr1.use_count() << endl;  
  
    return 0;  
}
```

```
Value ptr1: 100  
Pointers counter ptr1: 1  
Value ptr2: 100  
Pointers counter ptr1: 2  
After ptr2 deletion  
Pointers counter ptr1: 1
```

- ❑ `shared_ptr` дозволяє **декільком об'єктам** спільно володіти динамічною пам'яттю.
- ❑ Об'єкт буде знищено **лише тоді**, коли **всі** `shared_ptr`, що на нього вказують, знищаться або будуть переприсвоєні.

Приклад реалізації

```
struct Test {
    Test() { std::cout << "Test constructed\n"; }
    ~Test() { std::cout << "Test destroyed\n"; }
    void hello() { std::cout << "Hello!\n"; }
};

int main() {
    SharedPtr<Test> sp1(new Test); // ref_count = 1
    {
        SharedPtr<Test> sp2 = sp1; // ref_count = 2
        sp2->hello();
        std::cout << "Use count: " << sp2.use_count() << '\n';
    } // sp2 вийшов з області видимості, ref_count = 1

    std::cout << "Use count: " << sp1.use_count() << '\n';
} // sp1 видаляється, ref_count = 0 → delete Test
```

```
Test constructed
Hello!
Use count: 2
Use count: 1
Test destroyed
```

```
template<typename T>
class SharedPtr {
private:
    T* ptr; // вказівник на об'єкт
    int* ref_count; // лічильник посилянь

public:
    explicit SharedPtr(T* p = nullptr)
        : ptr(p), ref_count(new int(1)) {
    }

    SharedPtr(const SharedPtr& other)
        : ptr(other.ptr), ref_count(other.ref_count) {
        ++(*ref_count);
    }

    SharedPtr& operator=(const SharedPtr& other) {
        if (this != &other) {
            // Зменшити старий лічильник і, якщо треба, видалити
            release();

            // Копіювати з іншого об'єкта
            ptr = other.ptr;
            ref_count = other.ref_count;
            ++(*ref_count);
        }
        return *this;
    }

    ~SharedPtr() {
        release();
    }

    T& operator*() const { return *ptr; }
    T* operator->() const { return ptr; }
    int use_count() const { return *ref_count; }

private:
    void release() {
        if (--(*ref_count) == 0) {
            delete ptr;
            delete ref_count;
        }
    }
};
```

Приклад циклічної залежності

Циклічна залежність — це проблема, яка виникає при використанні `shared_ptr`, коли два (або більше) об'єкти зберігають **взаємні посилання один на одного** через `shared_ptr`. Внаслідок цього **лічильники посилань ніколи не стають нульовими**, тому пам'ять **не звільняється** — виникає **витік пам'яті**.

```
struct B; // forward declaration

struct A {
    std::shared_ptr<B> b_ptr;
    ~A() { std::cout << "A destroyed\n"; }
};

struct B {
    std::shared_ptr<A> a_ptr;
    ~B() { std::cout << "B destroyed\n"; }
};

int main() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();

    a->b_ptr = b;
    b->a_ptr = a;

    // Лічильники обох - 2, але програма закінчується,
    // і об'єкти НЕ знищуються, бо утримують один одного
}
```


weak_ptr

- ❑ `weak_ptr` — це допоміжний розумний вказівник, який **спостерігає** за об'єктом, яким володіє `shared_ptr`, але **не впливає** на його лічильник посилань.
- ❑ Використовується для **розриву циклічних посилань** між `shared_ptr`.
- ❑ `#include <memory>`

Синтаксис:

- ❑ `shared_ptr<T> sp = make_shared<T>(...);`
- ❑ `weak_ptr<T> wp = sp;`

Примітки:

- ❑ **Не збільшує** лічильник посилань `use_count()`.
- ❑ Щоб отримати доступ до об'єкта — використовують метод `.lock()`, який повертає `shared_ptr`.
- ❑ Можна перевірити, чи об'єкт ще існує, через `.expired()`.

Приклад weak_ptr

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    shared_ptr<int> shared = make_shared<int>(77);
    weak_ptr<int> weak = shared;

    cout << "shared.use_count(): " << shared.use_count() << endl; // Виведе: 1

    // Отримання доступу через weak_ptr
    if (auto locked = weak.lock()) {
        cout << "Значення: " << *locked << endl; // Виведе: 77
        cout << "locked.use_count(): " << locked.use_count() << endl; // Виведе: 2
    } else {
        cout << "Об'єкт вже знищено" << endl;
    }

    // shared_ptr зникає — об'єкт видаляється
    shared.reset();

    // Перевірка, чи weak_ptr ще валідний
    if (weak.expired()) {
        cout << "weak_ptr більше не дійсний" << endl;
    }

    return 0;
}
```

```
shared.use_count(): 1
Значення: 77
locked.use_count(): 2
weak_ptr більше не дійсний
```

- ❑ **weak_ptr** самостійно **не володіє** пам'яттю.
- ❑ **.lock()** повертає **shared_ptr**, якщо об'єкт ще існує.
- ❑ **.expired()** показує, чи об'єкт уже знищено

Приклад виправлення циклічної

```
#include <iostream>
#include <memory>

struct B; // наперед

struct A {
    std::shared_ptr<B> b_ptr;
    ~A() { std::cout << "A destroyed\n"; }
};

struct B {
    std::weak_ptr<A> a_ptr; // weak_ptr не заважає звільненню A
    ~B() { std::cout << "B destroyed\n"; }
};

int main() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();

    a->b_ptr = b;
    b->a_ptr = a;

    // Тепер цикл не заважає знищенню
}
```

A destroyed
B destroyed

Дякую