1. Определение указателя

Указатель - это переменная, которая хранит адрес другой переменной в памяти.

```
// Объявление указателя
тип_данных* имя_указателя;

// Примеры
int* ptr;       // указатель на int
double* dptr;   // указатель на double
char* cptr;     // указатель на char
```

Операторы:

- `&` - оператор взятия адреса
- `*` - оператор разыменования (доступ к значению по адресу)

```
int x = 10;
int* ptr = &x;   // ptr хранит адрес переменной x

cout << "Адрес x: " << ptr << endl;      // выведет адрес
cout << "Значение x: " << *ptr << endl;  // выведет 10
```

2. Операции с указателями

Основные операции:

- Присваивание
- Разыменование
- Сравнение
- Приведение типов

```
int a = 5, b = 10;
int* ptr1 = &a;
int* ptr2 = &b;

// Сравнение указателей
if (ptr1 == ptr2) {
    cout << "Указатели равны" << endl;
}
```

```
// Присваивание
ptr2 = ptr1;  // теперь оба указывают на a

// Изменение значения через указатель
*ptr1 = 20;
cout << "a = " << a << endl;  // выведет 20
```

3. Арифметика указателей

Указатели можно увеличивать/уменьшать, но с учетом размера типа данных.

```
int arr[5] = {10, 20, 30, 40, 50};
int* ptr = arr;  // ptr указывает на первый элемент

cout << *ptr << endl;  // 10
ptr++;                 // перемещаемся на следующий int (4 байта)
cout << *ptr << endl;  // 20

ptr += 2;              // перемещаемся на два элемента вперед
cout << *ptr << endl;  // 40

ptr--;                 // перемещаемся на один элемент назад
cout << *ptr << endl;  // 30
```

Разность указателей:

```
int* ptr1 = &arr[0];
int* ptr2 = &arr[3];
cout << "Разность: " << (ptr2 - ptr1) << endl;  // выведет 3
```

4. Константы и указатели

Есть несколько вариантов сочетания const с указателями:

```
int x = 10;
int y = 20;

// 1. Указатель на константу - данные нельзя менять через указатель
const int* ptr1 = &x;
// *ptr1 = 30;  // ОШИБКА! Данные константные
```

```
ptr1 = &y;      // OK - сам указатель можно менять

// 2. Константный указатель - нельзя менять адрес
int* const ptr2 = &x;
*ptr2 = 30;     // OK - данные можно менять
// ptr2 = &y;   // ОШИБКА! Указатель константный

// 3. Константный указатель на константу
const int* const ptr3 = &x;
// *ptr3 = 30;  // ОШИБКА
// ptr3 = &y;   // ОШИБКА
```

5. Массивы и указатели

Массивы и указатели тесно связаны: имя массива - это указатель на его первый элемент.

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

// Эквивалентные способы доступа к элементам
cout << arr[2] << endl;     // 3
cout << *(arr + 2) << endl; // 3

// Указатель можно использовать как массив
int* ptr = arr;
cout << ptr[1] << endl;     // 2

// Передача массива в функцию
void printArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
}
```

6. Динамическая память и smart-указатели

Обычные указатели (ручное управление памятью)

```
// Выделение памяти
```

```cpp
int* ptr = new int;
*ptr = 100;

// Выделение массива
int* arr = new int[5];
for (int i = 0; i < 5; i++) {
    arr[i] = i * 10;
}

// Освобождение памяти
delete ptr;      // для одиночного объекта
delete[] arr;    // для массива
```

Умные указатели (automatic memory management)

1. unique_ptr - эксклюзивное владение (C++11)

```cpp
#include <memory>

std::unique_ptr<int> ptr1 = std::make_unique<int>(100);
std::unique_ptr<int[]> arr1 = std::make_unique<int[]>(5);

// Нельзя копировать, можно только перемещать
std::unique_ptr<int> ptr2 = std::move(ptr1);
```

2. shared_ptr - разделяемое владение с подсчетом ссылок (C++11)

```cpp
std::shared_ptr<int> ptr1 = std::make_shared<int>(100);
std::shared_ptr<int> ptr2 = ptr1;  // копирование разрешено

cout << "Количество ссылок: " << ptr1.use_count() << endl; // 2
```

3. weak_ptr - слабая ссылка без увеличения счетчика (C++11)

```cpp
std::shared_ptr<int> shared = std::make_shared<int>(100);
std::weak_ptr<int> weak = shared;

// Для доступа нужно преобразовать в shared_ptr
if (auto temp = weak.lock()) {
```

```
    cout << *temp << endl;
}
```

Практический пример

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

int main() {
    // Обычный указатель
    int* rawPtr = new int(42);
    std::cout << "Raw pointer: " << *rawPtr << std::endl;
    delete rawPtr;

    // Умный указатель
    std::unique_ptr<int> smartPtr = std::make_unique<int>(84);
    std::cout << "Smart pointer: " << *smartPtr << std::endl;

    // Автоматическое освобождение памяти!
    return 0;
}
```

Рекомендация: всегда используйте умные указатели вместо обычных, когда это возможно. Они предотвращают утечки памяти и делают код безопаснее.

Дополнительная информация

Размер указателей

Пояснение: Размер указателя в первую очередь определяется архитектурой целевой платформы (например, 32-битная vs 64-битная), а не типом данных, на который он указывает. Указатель — это просто адрес в памяти, поэтому размер всех типизированных указателей (`int*`, `char*`) и `void*` на одной платформе одинаков. Однако указатели на функции могут иметь иной размер и представление, чем указатели на данные, что зависит от платформы и компилятора.

Пример:

```
#include <iostream>
int main() {
    int* ptr_int;
```

```
    char* ptr_char;
    void* ptr_void;
    std::cout << "sizeof(int*): " << sizeof(ptr_int) << '\n';
    std::cout << "sizeof(char*): " << sizeof(ptr_char) << '\n';
    std::cout << "sizeof(void*): " << sizeof(ptr_void) << '\n';
    // На большинстве современных ПК все выводы будут одинаковыми (8 байт для x86-64)
}
```

Инвалидация указателей

Пояснение: Указатель становится недействительным (dangling), когда память, на которую он указывает, перестает быть доступной. Это происходит при освобождении динамической памяти (`delete`), когда объект в стеке выходит из области видимости, или когда умный указатель, владеющий объектом, освобождает его. Модификация константного объекта через неконстантный указатель — это попытка нарушить константность, которая приводит к неопределенному поведению, но не обязательно инвалидирует сам указатель.

Пример:

```
int* create_int() {
    int x = 42;
    return &x; // Возвращаем указатель на локальную переменную -> dangling pointer
}
int main() {
    int* p1 = new int(10);
    delete p1; // p1 теперь dangling

    int* p2 = create_int(); // p2 dangling сразу после вызова

    auto sptr = std::make_shared<int>(20);
    int* p3 = sptr.get();
    sptr.reset(); // Освобождаем память, p3 становится dangling
}
```

Операции с void*

Пояснение: Указатель `void*` — это указатель на сырую память без информации о типе. Его нельзя разыменовать, так как компилятор не знает, как интерпретировать данные по этому адресу. Для него также не определена арифметика указателей (например, `++`),

потому что компилятор не знает размер целевого типа для вычисления смещения. Однако его можно сравнивать с другими указателями (включая nullptr) и явно приводить к другим типам указателей.

Пример:

```
int x = 10;
void* vptr = &x;

// *vptr = 20; // Ошибка: нельзя разыменовать void*
// vptr++;     // Ошибка: арифметика указателей для void* не определена

if (vptr != nullptr) { // Сравнение разрешено
    int* iptr = static_cast<int*>(vptr); // Приведение разрешено
    *iptr = 20; // Теперь можно работать через типизированный указатель
}
```

Указатели на константный массив

Пояснение: Объявление указателей на массивы требует внимания к скобкам. const int* arr[10] — это массив из 10 указателей на константные int. Чтобы объявить указатель на массив, нужны скобки: int (*arr)[10] — указатель на массив из 10 int. Добавление const слева (const int (*arr)[10]) делает элементы массива константными. Добавление const после * (int (* const arr)[10]) делает сам указатель константным.

Пример:

```
int main() {
    int array[10] = {};
    const int const_array[10] = {};

    // Указатель на массив из 10 int
    int (*ptr_to_array)[10] = &array;

    // Указатель на массив из 10 const int
    const int (*ptr_to_const_array)[10] = &const_array; // OK
    // ptr_to_const_array = &array; // Также OK, ибо неконст. int можно трактовать как const
    // (*ptr_to_const_array)[0] = 1; // Ошибка: элементы константны
```

```
    // Константный указатель на массив из 10 int
    int (* const const_ptr_to_array)[10] = &array;
    // const_ptr_to_array = &array; // Ошибка: сам указатель константный
    (*const_ptr_to_array)[0] = 1; // OK: элементы не константны
}
```

Арифметика указателей

Пояснение: Арифметика указателей работает не с байтами, а с элементами. Когда вы прибавляете n к указателю типа T*, компилятор вычисляет новый адрес как текущий адрес + n * sizeof(T). Разность двух указателей возвращает количество *элементов* типа T* между ними, а не байт. Сравнение указателей разных типов возможно только после явного приведения kvoid*или совместимым типам. Вычитаниеnullptr` или любого другого невалидного указателя приводит к неопределенному поведению.

Пример:

```
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int* ptr1 = &arr[0];
    int* ptr2 = &arr[3];

    ptr1 = ptr1 + 2; // Теперь ptr1 указывает на arr[2]
    std::cout << *ptr1 << std::endl; // Выведет 30

    ptrdiff_t diff = ptr2 - ptr1; // diff = 1 (элемент), а не sizeof(int) байт
    std::cout << diff << std::endl;

    char* cptr = reinterpret_cast<char*>(ptr1);
    cptr++; // Инкремент на 1 байт (размер char)
}
```

Доступ к элементам массива

Пояснение: В C++ имя массива в выражениях (кроме операторов sizeof и &) неявно преобразуется (decay) в указатель на свой первый элемент. Поэтому обращение к элементу массива arr[i] синтаксически эквивалентно *(arr + i). Из-за коммутативности сложения это также эквивалентно *(i + arr), что bizarrely позволяет запись i[arr]. Выражение *arr + i эквивалентно arr[0] + i, что является арифметикой значений, а не

указателей.

Пример:

```
int main() {
    int arr[3] = {100, 200, 300};

    // Все эти строки выводят один и тот же элемент (200)
    std::cout << arr[1] << std::endl;
    std::cout << *(arr + 1) << std::endl;
    std::cout << 1[arr] << std::endl; // Неочевидная, но рабочая запись

    std::cout << *arr + 1 << std::endl; // Выведет 100 + 1 = 101
}
```

Гарантии std::unique_ptr

Пояснение: std::unique_ptr обеспечивает исключительное владение динамически выделенным объектом и гарантирует его автоматическое уничтожение (через delete или заданный Deleter) при выходе из области видимости умного указателя. Его нельзя скопировать (только переместить), что и обеспечивает уникальность владения. Специализация std::unique_ptr<T[]> корректно обрабатывает массивы (вызывает delete[]). std::unique_ptr не гарантирует потокобезопасность операций с самим указателем; доступ к нему из разных потоков требует синхронизации.

Пример:

```
#include <memory>
void func() {
    // Уникальное владение одиночным объектом
    std::unique_ptr<int> uptr1(new int(5));
    // auto uptr1 = std::make_unique<int>(5); // Предпочтительнее

    // Уникальное владение массивом
    std::unique_ptr<int[]> uptr_arr(new int[10]{});
    // auto uptr_arr = std::make_unique<int[]>(10); // Предпочтительнее
    uptr_arr[0] = 42;

    // std::unique_ptr<int> uptr_copy = uptr1; // Ошибка: копирование запрещено
```

```
    std::unique_ptr<int> uptr_moved = std::move(uptr1); // OK: перемещение
} // Память автоматически освобождается здесь для uptr_moved и uptr_arr
```

Счетчик ссылок std::shared_ptr

Пояснение: std::shared_ptr реализует подсчет ссылок для разделяемого владения объектом. Счетчик увеличивается при создании новой копии shared_ptr (конструктор копирования, оператор присваивания). При перемещении shared_ptr счетчик не изменяется, так как владение передается от одного указателя другому. Вызов reset() уменьшает счетчик, так как текущий shared_ptr освобождает владение. Создание std::weak_ptr из shared_ptr не изменяет счетчик владения, но может увеличивать отдельный "слабый" счетчик.

Пример:

```
#include <iostream>
#include <memory>
int main() {
    auto sptr1 = std::make_shared<int>(42);
    std::cout << "Count after sptr1: " << sptr1.use_count() << std::endl; // 1

    { // Новая область видимости
        std::shared_ptr<int> sptr2 = sptr1; // Копирование -> счетчик +1
        std::cout << "Count after sptr2 copy: " << sptr1.use_count() << std::endl; // 2

        std::shared_ptr<int> sptr3 = std::move(sptr2); // Перемещение -> счетчик не меняется
        std::cout << "Count after sptr3 move: " << sptr1.use_count() << std::endl; // 2
        // sptr2 теперь nullptr

        sptr3.reset(); // Явный вызов reset() -> счетчик -1
        std::cout << "Count after sptr3 reset: " << sptr1.use_count() << std::endl; // 1
    } // sptr3 выходит из области видимости, но он уже reset, поэтому счетчик не меняется

    std::weak_ptr<int> wptr = sptr1; // Создание weak_ptr не меняет use_count
    std::cout << "Count after weak_ptr creation: " << sptr1.use_count() << std::endl; // 1
}
```

Dangling pointer

Пояснение: Dangling pointer ("висячий указатель") — это указатель, который продолжает существовать после того, как память, на которую он указывал, была освобождена или перестала существовать. Это может произойти из-за освобождения динамической памяти, выхода локальной переменной из области видимости или даже изменения базового адреса (например, при переаллокации вектора `std::vector`). Неинициализированный указатель содержит случайный адрес и это другая проблема. Указатель на константный объект корректен, если сам объект still exists.

Пример:

```
#include <vector>
int* create_dangling() {
    int local = 42;
    return &local; // Возврат указателя на локальную переменную -> dangling
}
int main() {
    // 1. Указатель на освобожденную память
    int* dyn_ptr = new int(10);
    delete dyn_ptr;
    // dyn_ptr теперь dangling

    // 2. Указатель на вышедший из области видимости объект
    int* dangling_func_ptr = create_dangling(); // dangling

    // 3. Указатель, инвалидированный сдвигом базового адреса
    std::vector<int> vec = {1, 2, 3};
    int* elem_ptr = &vec[0];
    vec.push_back(4); // push_back может вызвать переаллокацию всего вектора
    // elem_ptr теперь dangling, если произошла переаллокация
}
```

reinterpret_cast для указателей

Пояснение: `reinterpret_cast` — это наиболее мощное и опасное приведение типов в C++. При преобразовании указателей он просто интерпретирует битовое представление адреса исходного указателя как адрес целевого типа. Он не выполняет никаких проверок выравнивания или семантической корректности. Такое преобразование может легко нарушать strict aliasing rules, что ведет к неопределенному поведению. Он может

конвертировать между любыми типами указателей, но результат может быть нечитаемым, если типы несвязаны.

Пример:

```
#include <iostream>
int main() {
    int x = 0x41424344; // Пример целого числа
    int* int_ptr = &x;

    // Преобразование int* в char* для доступа к отдельным байтам
    char* char_ptr = reinterpret_cast<char*>(int_ptr);
    for (size_t i = 0; i < sizeof(x); ++i) {
        std::cout << char_ptr[i] << " "; // Может вывести 'D', 'C', 'B', 'A' (зависит от endianness)
    }

    // Преобразование между несвязанными типами (UB если разыменовать)
    double* double_ptr = reinterpret_cast<double*>(int_ptr);
    // *double_ptr = 3.14; // Неопределенное поведение: нарушение strict aliasing

    // Сохранение и восстановление адреса через void* (часто безопаснее)
    void* void_ptr = int_ptr; // Неявное преобразование в void*
    int* int_ptr_again = static_cast<int*>(void_ptr); // Обратное преобразование
}
```

Связь массивов и указателей

Пояснение: Имя массива в большинстве контекстов преобразуется (decay) в указатель на свой первый элемент. Однако есть исключения: оператор `sizeof` возвращает размер всего массива в байтах, а оператор `&` возвращает адрес всего массива (тип `int(*)[N]`), который численно равен адресу первого элемента, но имеет другой тип. Многомерные массивы в C++ хранятся по строкам (row-major order): сначала первая строка, затем вторая и т.д.

Пример:

```
#include <iostream>
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
```

```
    // arr decay к указателю на первый элемент в этом выражении
    int* ptr = arr;
    std::cout << *ptr << std::endl; // 1

    // Исключения для decay:
    std::cout << "Sizeof array: " << sizeof(arr) << std::endl; // 5 * sizeof(int)
    int (*ptr_to_array)[5] = &arr; // Указатель на весь массив

    // Многомерный массив (row-major)
    int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
    // В памяти: 1, 2, 3, 4, 5, 6
    int* first_element = &matrix[0][0];
    std::cout << *(first_element + 3) << std::endl; // Доступ к элементу [1][0] -> 4
}
```

const и указатели

Пояснение: Ключевое слово `const` при объявлении указателей можно применять к самому указателю (делая его константным) или к данным, на которые он указывает (делая данные доступными только для чтения). `const int* p` и `int const* p` эквивалентны — указатель на константные данные. `int* const p` — константный указатель на неконстантные данные. `const int* const p` — константный указатель на константные данные. `const_cast` может убрать константность, но изменение действительно константного объекта через такой указатель — это неопределенное поведение.

Пример:

```
int main() {
    int a = 10;
    const int b = 20;

    // Указатель на константные данные
    const int* p1 = &a;
    // *p1 = 15; // Ошибка: данные константны
    p1 = &b;     // OK: указатель не константен

    // Константный указатель
```

```
    int* const p2 = &a;
    *p2 = 15;    // OK: данные не константны
    // p2 = &b;  // Ошибка: указатель константен

    // Константный указатель на константные данные
    const int* const p3 = &b;
    // *p3 = 25; // Ошибка: данные константны
    // p3 = &a;  // Ошибка: указатель константен

    // Использование const_cast (ОПАСНО!)
    const int* p4 = &b;
    int* p5 = const_cast<int*>(p4);
    // *p5 = 30; // Неопределенное поведение, т.к. b - действительно константа
}
```

Полезность std::weak_ptr

Пояснение: std::weak_ptr решает две основные проблемы std::shared_ptr: циклические ссылки и наблюдение без владения. weak_ptr не увеличивает счетчик ссылок shared_ptr, поэтому не препятствует уничтожению объекта, когда все shared_ptr освобождаются. Это позволяет безопасно наблюдать за объектом и проверять его существование (через lock()), не продлевая его время жизни. Он не управляет временем жизни объекта и не освобождает память.

Пример:

```
#include <iostream>
#include <memory>
struct Node {
    // std::shared_ptr<Node> next; // Циклическая ссылка -> утечка памяти
    std::weak_ptr<Node> next; // Решение: weak_ptr не увеличивает счетчик
    ~Node() { std::cout << "Node destroyed\n"; }
};
int main() {
    auto node1 = std::make_shared<Node>();
    auto node2 = std::make_shared<Node>();
    node1->next = node2; // Присваивание weak_ptr из shared_ptr
    node2->next = node1;
```

```
    // Проверка существования объекта и получение доступа
    if (auto shared_next = node1->next.lock()) {
        std::cout << "Node1's next still exists\n";
        // Можно работать с shared_next как с shared_ptr
    } else {
        std::cout << "Node1's next has been destroyed\n";
    }
} // Память корректно освобождается благодаря weak_ptr
```

UB операций с указателями

Пояснение: Неопределенное поведение (UB) возникает при операциях с указателями, которые нарушают правила языка. Разыменование `nullptr` или невалидного указателя — классическое UB. Выход за границы массива при арифметике или индексации — UB. Сравнение указателей, относящихся к разным массивам (или объектам), с помощью операторов `<`, `<=`, `>`, `>=` — UB, хотя на практике оно часто работает ожидаемо, стандарт этого не гарантирует. Преобразование integer-to-pointer и обратно — implementation-defined, но не UB само по себе.

Пример:

```
int main() {
    int* p1 = nullptr;
    // *p1 = 42; // UB: разыменование nullptr

    int arr[5] = {};
    int* p2 = &arr[0];
    p2 += 10;   // UB: выход за границы массива
    // int val = *p2; // UB

    int another_arr[5];
    int* p3 = &another_arr[0];
    // bool comparison = p2 < p3; // UB: сравнение указателей из разных массивов

    // Преобразование int в указатель (implementation-defined, часто для embedded)
    uintptr_t addr = 0x1000;
    int* p4 = reinterpret_cast<int*>(addr); // Не UB, но опасно
```

}

Указатели на функции

Пояснение: Указатель на функцию хранит адрес функции, которую можно вызвать later. Тип указателя полностью определяется типом возвращаемого значения и типами аргументов функции. Указатели на функции несовместимы с лямбда-выражениями, которые захватывают переменные (имеют состояние), но совместимы с лямбдами без захвата, которые могут быть неявно преобразованы в указатель на функцию. Ссылки на функции (`void(&)(int)`) — это псевдонимы для существующих функций, в то время как указатели (`void(*)(int)`) — это переменные, которые можно переназначать. Вызов функции через указатель синтаксически похож на прямой вызов.

Пример:

```
#include <iostream>
void print_hello(int times) {
    for (int i = 0; i < times; ++i) std::cout << "Hello! ";
    std::cout << std::endl;
}
int add(int a, int b) { return a + b; }
int main() {
    // Объявление и инициализация указателя на функцию
    void (*func_ptr)(int) = &print_hello;
    // void (*func_ptr)(int) = print_hello; // & опционально

    // Вызов функции через указатель
    func_ptr(3); // Выведет "Hello! Hello! Hello! "

    // Переназначение указателя (нельзя для ссылки на функцию)
    int (*math_ptr)(int, int) = &add;
    std::cout << math_ptr(5, 3) << std::endl; // 8

    // Лямбда без захвата может быть преобразована
    auto lambda = [](int x) { std::cout << "Lambda: " << x << std::endl; };
    void (*lambda_ptr)(int) = lambda; // OK, т.к. нет захвата
    lambda_ptr(42);
}
```

Динамическое выделение массива

Пояснение: Для динамического выделения массива в C++ предпочтительнее использовать `std::vector` или умные указатели. Однако если нужен нативный массив, оператор `new[]` позволяет инициализацию: `new int[10]()` инициализирует все элементы значением по умолчанию (0 для `int`), а `new int[10]{1,2}` использует список инициализации (первые два элемента 1 и 2, остальные 0). `std::make_unique<int[]>(10)` создает `unique_ptr` на массив и value-initializes элементы. `malloc` лишь выделяет "сырую" память без вызова конструкторов и инициализации.

Пример:

```
#include <memory>
#include <cstdlib>
int main() {
    // Выделение и инициализация значением по умолчанию (0)
    int* arr1 = new int[10]();
    delete[] arr1;

    // Выделение и инициализация списком
    int* arr2 = new int[5]{1, 2, 3}; // [1, 2, 3, 0, 0]
    delete[] arr2;

    // Предпочтительный способ: умный указатель на массив
    std::unique_ptr<int[]> arr3 = std::make_unique<int[]>(5); // Все элементы 0
    arr3[0] = 10;

    // malloc — нет инициализации, не вызывает конструкторы
    int* arr4 = static_cast<int*>(malloc(5 * sizeof(int)));
    free(arr4);
}
```

Strict aliasing

Пояснение: Правило strict aliasing является допущением компилятора, что указатели разных типов не ссылаются на одну и ту же область памяти (за исключением нескольких разрешенных случаев). Это позволяет проводить агрессивные оптимизации. Нарушение этого правила (доступ к объекту через указатель несовместимого типа) ведет к неопределенному поведению. Явные исключения из этого правила — это указатели

типа char*, unsigned char* и std::byte*, которые могут использоваться для доступа к представлению любого объекта в памяти.

Пример:

```
#include <cstring>
int main() {
    int x = 0x11223344;

    // Нарушение strict aliasing (UB)
    float* fptr = reinterpret_cast<float*>(&x);
    // float value = *fptr; // Неопределенное поведение

    // Законный способ доступа к object representation через char*
    char* cptr = reinterpret_cast<char*>(&x);
    for (size_t i = 0; i < sizeof(x); ++i) {
        std::cout << static_cast<int>(cptr[i]) << " "; // Байтовое представление
    }

    // Законный способ: использование std::memcpy (копирует байты)
    float y;
    std::memcpy(&y, &x, sizeof(x)); // OK
    // Но значение y может быть некорректным float (но это не UB кода доступа)
}
```

Проверка указателей в constexpr

Пояснение: Контекст constexpr требует, чтобы вычисления выполнялись на этапе компиляции. Это накладывает ограничения на операции с указателями. В constexpr можно работать с адресами статических объектов (глобальных, статических переменных), так как их адреса известны компилятору. Можно выполнять арифметику указателей в пределах одного массива и разыменовывать валидные указатели. Однако reinterpret_cast и многие низкоуровневые преобразования указателей не допускаются в constexpr, так как их результат не может быть вычислен во время компиляции.

Пример:

```
constexpr int global = 42; // Статическая продолжительность хранения
constexpr int* get_global_ptr() {
```

```
    return const_cast<int*>(&global); // Можно взять адрес
}
constexpr int access_array() {
    int arr[5] = {1, 2, 3, 4, 5};
    int* ptr = &arr[0];
    ptr += 2;       // Арифметика указателей в пределах массива OK
    return *ptr;    // Разыменование OK -> вернет 3
}
// constexpr int invalid_ops() {
//     int x;
//     return reinterpret_cast<intptr_t>(&x); // Ошибка: reinterpret_cast в constexpr
// }
int main() {
    constexpr int* p = get_global_ptr();
    constexpr int val = access_array();
    static_assert(val == 3, "");
}
```

std::atomic<T*>

Пояснение: std::atomic<T*> — это специализация шаблона std::atomic для указателей. Она гарантирует, что операции чтения, записи и сравнения с обменом (read-modify-write) над указателем будут атомарными, то есть неделимыми с точки зрения других потоков. Эта специализация поддерживает арифметику указателей (fetch_add, fetch_sub) как атомарные операции. Стандарт рекомендует, чтобы специализации std::atomic для указателей были lock-free (реализованы на уровне процессорных инструкций), что обычно выполняется.

Пример:

```
#include <atomic>
#include <iostream>
#include <thread>
std::atomic<int*> atomic_ptr;
int data[100];
void writer() {
    for (int i = 0; i < 100; ++i) {
        // Атомарно изменяем указатель
```

```
        atomic_ptr.store(&data[i], std::memory_order_release);
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
}
void reader() {
    int* local_ptr;
    while ((local_ptr = atomic_ptr.load(std::memory_order_acquire)) < &data[99]) {
        std::cout << "Reader sees: " << *local_ptr << std::endl;
    }
}
int main() {
    atomic_ptr = &data[0];
    std::thread t1(writer);
    std::thread t2(reader);
    t1.join();
    t2.join();
    // Атомарная арифметика указателей
    int* old = atomic_ptr.fetch_add(1); // Атомарно делает atomic_ptr = atomic_ptr + 1
}
```

Ошибки std::shared_ptr

Пояснение: `std::shared_ptr` помогает избежать классических ошибок ручного управления памятью. Он предотвращает двойное освобождение (double free), так как освобождает память ровно один раз, когда счетчик ссылок обнуляется. Он предотвращает разыменование освобожденной памяти (use-after-free), так как объект уничтожается только когда на него не осталось `shared_ptr`. Он делает явным контроль владения — только копирование `shared_ptr` увеличивает счетчик. Однако он *не* предотвращает утечки памяти из-за циклических ссылок (для этого нужен `weak_ptr`).

Пример:

```
#include <memory>
void manual_double_free() {
    int* raw_ptr = new int(10);
    delete raw_ptr;
    // ... later, maybe in another part of code
    // delete raw_ptr; // Катастрофа: double free
```

```
}
void shared_ptr_prevents_double_free() {
    auto sptr1 = std::make_shared<int>(10);
    {
        auto sptr2 = sptr1; // Счетчик становится 2
    } // sptr2 уничтожается, счетчик становится 1
    // ...
} // sptr1 уничтожается, счетчик становится 0, память освобождается ровно 1 раз

void use_after_free() {
    // int* raw_ptr = new int(10);
    // delete raw_ptr;
    // int value = *raw_ptr; // UB: use-after-free

    auto sptr = std::make_shared<int>(10);
    // Множество копий shared_ptr
    // ...
    int value = *sptr; // Всегда безопасно, если хотя бы один shared_ptr жив
}

// shared_ptr НЕ предотвращает циклические ссылки (утечка памяти):
struct BadNode {
    std::shared_ptr<BadNode> next;
};
void cyclic_leak() {
    auto node1 = std::make_shared<BadNode>();
    auto node2 = std::make_shared<BadNode>();
    node1->next = node2;
    node2->next = node1; // Цикл -> счетчики ссылок никогда не станут 0
} // Память утекает
```