Лекция 2. CV-квалификаторы. Ссылки. Указатели

1. CV-квалификаторы
   - `const`
   - `volatile`
   - `const volatile`
   - Оператор `const_cast`
2. Ссылка (*reference*)
   - Оператор доступа `[]`
   - Ссылка на константу
   - Ссылка на массив
   - Итерирование по контейнеру
3. Указатель (*pointer*)
   - Оператор взятия адреса `&` (*address-of*)
   - Использование амперсанда `&`
   - Получение адреса `std::addressof()` (*C++11*)
   - Оператор разыменования `*` (*indirection*)
   - Использование звездочки `*`
   - Стиль кода амперсанда и звездочки
   - Совместное использование амперсанда и звездочки
   - Константность указателя
   - Нулевой указатель `nullptr`
   - Арифметика указателей
   - Сравнение указателей
   - Расстояние между указателями
   - Передача массива через указатели
   - Ссылка на указатель
   - Указатель на указатель
   - Указатель на `void`
   - Оператор `reinterpret_cast`
   - Указатель на функцию
   - Преобразование *function to function pointer*
   - Сложные объявления с указателями
4. Передача аргументов в функцию
   - Продление жизни ссылки на константу
   - Почему нет продления жизни не константной ссылки
   - Висячая ссылка (*dangling reference*)
   - Невалидный указатель (*invalid pointer*)v

CV-квалификаторы

В языке C++ существуют квалификаторы типа.

CV означает `const` `volatile`

Квалификаторы могут быть применены к типам переменных и методам (функциям внутри класса)

`const`

Квалификатор типа отвечающий за константность.

Означает, что объект данного типа нельзя изменять

Если тип объявлен с данным квалификатором, то компилятор не разрешает пользователю изменять данную сущность. Что позволяет находить ошибки в коде на этапе компиляции (например, при попытке изменить переменную с квалификатором `const`)

Все глобальные переменные рекомендуется объявлять константными.

Можно написать как **ДО** типа так и **ПОСЛЕ**:

```
const int a = 0;
int const b = 0;
```

- **НО** принято писать **ДО** типа

Невозможно делать операции с переменной данного типа, подразумевающие изменение (инкремент/декремент, присваивания, ...)

`volatile`

Квалификатор типа отвечающий за изменчивость.

Означает, что объект данного типа может неожиданно измениться
(из-за действий в другом потоке, по другим причинам) и его можно изменять пользователю.

Сообщает компилятору, что переменная видима извне, а значит не нужно оптимизировать код, связанный с данной переменной.

```
volatile int a = 0;
int volatile b = 0;
```

`const volatile`

Можно использовать оба квалификатора.

Означает, что объект нельзя изменять пользователю, но он может неожиданно измениться.

Оператор `const_cast`

`const_cast` - оператор приведения типа, который необходим для добавления или удаления квалификаторов `const`, `volatile`.

Синтаксис: `const_cast<new_type>(<expression>)`

- `new_type` - тип, к которому необходимо привести
- `<expression>` - выражение, результат которого необходимо привести к указанному типу
- Используется в *legacy* коде, так как в языке C нет `const`
- Возможно удалить `const` с объектов, которые не были объявлены `const` но стали константными в результате передачи в функцию.

```
void change_const_value(const int& value) {
    int& mutable_value = const_cast<int&>(value);
    mutable_value = 100; // только если оригинал не const
}

int main() {
    int original = 42;
    std::cout << "before const_cast: " << original << std::endl;
    change_const_value(original);
    std::cout << "after const_cast: " << original << std::endl;
}
```

Ссылка (*reference*)

Ссылка представляет собой псевдоним для переменной.

Ссылка позволяет использовать объект избегая копирования, ссылка не хранит значение.

Для создания ссылки используется амперсанд `&` после имени типа.

Синтаксис: `<type>& <alias_name> = <exist_name>;`

- `<alias_name>` - имя ссылки
- `<exist_name>` - имя существующей переменной

Представляет собой новый составной тип: ссылка на тип.

Особенности:

- нельзя создать ссылку без инициализации
- ссылка может быть связанна с переменной один раз
- ссылки на ссылку не существует

Ссылка позволяет изменять исходный объект: `<ref_name> = <value>`

```
int x = 0;
int& y = x;
y = 1;
std::cout << x; // 1
```

```
int x = 0, z = 5;
int& y = x;
y = z;          // change x, not change ref for y
std::cout << x; // 5
```

Оператор доступа `[]`

Оператор `[]` возвращает ссылку на объект:

```
int arr[] = {1, 2, 3, 4, 5};
int& value = arr[2];
value = 0;  // arr = {1, 2, 0, 4, 5}
arr[3] = 8; // arr = {1, 2, 0, 8, 5}
```

Ссылка на константу

Ссылку на константу часто называют константной ссылкой.

При добавлении константности к ссылке, нельзя поменять переменную через ссылку

```
int x = 0;
const int& ref = x;
ref = 1; // compile error
```

На константную переменную может ссылаться только ссылка на константу:

```
const int value = 100;
int& ref = value;        // compile error
const int& cref = value; // ok
```

На литерал нельзя сделать ссылку, потому что литерал это константа времени компиляции. Но можно сделать константную ссылку на литерал, поскольку в языке принято продление жизни константных ссылок (*lifetime expansion*)

```
int& ref = 10;        // compile error
const int& cref = 10; // ok
```

- В этот момент создается объект со значением литерала, на который создается ссылка

Ссылка на массив

Ссылку на массив можно создать следующим образом:

```
int arr[10];
int (&ref_a)[10] = arr;
int matrix[3][4];
int (&ref_m)[3][4] = matrix;
```

- Придется явно указать размерность.
- При передачи в функцию сохраняется воможность итерироваться по массиву в цикле *range-based for*

Итерирование по контейнеру

Для тяжелых типов важно правильно итерироваться по контейнерам и стандартным массивам.

Спецификатор типа `auto` срезает тип, а именно ссылку и константность.
Поэтому его нужно уточнять с помощью `const` и `&`.

Копирование не подходит для тяжелых объектов (`std::string` - строка C++,
может быть тяжелым объектом).

```
std::vector<std::string> book_names;
// some code to fill book_names...
for (auto book_name : book_names) {
    // do something...
}
```

Если необходимо изменять объект в цикле, то используется ссылка `auto&`.
Для тяжелых объектов позволяет избежать копирования и значительно ускорить итерирование по контейнеру.

```
std::vector<std::string> book_names;
// some code to fill book_names...
for (auto& book_name : book_names) {
    // change book_names strings through ref book_name...
}
```

Если **НЕТ** необходимости менять объект используется `const auto&`

```
std::vector<std::string> book_names;
// some code to fill book_names...
for (const auto& book_name : book_names) {
    // do something without changing of book_name...
}
```

ВАЖНО для легких типов (фундаментальных, перечислений, типов до 8-16 байт), если нет необходимости изменять объект, использовать не `const auto&`, а копирование через `auto`. Это будет быстрее.

```
std::vector<int> values;
// some code to fill values...
for (auto value : values) {
    std::cout << value << std::endl;
}
```

Указатель (*pointer*)

Указатель это тип, который позволяет хранить адрес другого объекта в памяти.

Синтаксис: `<type>* <name> = <address>;`

- `*` - обозначение указателя
- `<type>` - тип объекта, на который указывает указатель
- `<name>` - имя для указателя, иногда начинается с `ptr_`
- `<address>` - адрес объекта в памяти на который указывает указатель

```
int x = 0xBADC0FFE;
int* ptr_x = &x;
std::cout << x << std::endl;
std::cout << ptr_x << std::endl;
std::cout << *ptr_x << std::endl;
```

Особенности:

- указатели могут быть не инициализированы (но так не стоит делать)
- можно поменять объект на который указывает указатель
- можно создавать указатель на указатель

Оператор взятия адреса `&` (*address-of*)

- Унарный оператор, который используется перед именем сущности для взятия адреса.

```cpp
int x = 42;
std::cout << &x << std::endl;  // address-of x
const double pi = 3.1415926;
std::cout << &pi << std::endl; // address-of pi
int& y = x;
std::cout << &y << std::endl;  // address-of x
```

Использование амперсанда `&`

- Символ используется в разных контекстах:

```cpp
int x = 0;
// Побитовое И
auto y = x & 0xBADC0FFE;
// Объявление ссылки
int& ref = x;
// Взятие адреса
auto addr = &x;
```

Получение адреса `std::addressof()` (*C++11*)

В стандартной библиотеке языка C++ в заголовочном файле `<memory>` присутствует шаблонная функция `std::addressof`, которая позволяет взять правильный адрес объекта, даже если перегружен оператор амперсанд `&`.

Функция возвращает указатель на тип аргумента (адрес аргумента).

```cpp
int x = 10;
std::cout << std::addressof(x) << std::cou;
```

Оператор разыменования `*` (*indirection*)

- Унарный оператор, позволяющий по адресу получить доступ к объекту, который лежит по данному адресу. Называется `indirection` или `dereference`.

```cpp
int x = 0xBADC0FFE;
int* ptr_x = &x;
std::cout << *ptr_x << std::endl; // dereference ptr_x
int y = *ptr_x; // dereference ptr_x
```

Как правило, возвращает ссылку на объект, поэтому можно изменять значение:

```cpp
int x = 5;
int* ptr = &x;
*ptr = 6;
std::cout << x << std::endl; // 6
```

Использование звездочки `*`

- Символ используется в разных контекстах:

```cpp
int x = 42;
// Умножение
auto mul = x * 42;
// Объявление указателя
int* ptr = &x;
// Разыменование указателя
auto y = *ptr; // copy
auto& ref = *ptr; // ref
```

Стиль кода амперсанда и звездочки

Это не обсуждается, используем первый вариант!

```cpp
int* ptr = &x;
int& ref = x;
```

Так тоже можно:

```cpp
int * ptr = &x;
int & ref = x;
```

Legacy языка C:

```cpp
int *ptr = &x;
int &ref = x;
```

Может пригодиться при объявлении цепочки указателей

```cpp
int *p = nullptr, *p1 = nullptr, *p3 = nullptr;
```

- но нормальный *style guide* просит объявлять переменные в отдельной строке

Совместное использование амперсанда и звездочки

Операторы разыменования и взятия адреса можно использовать совместно

```
int x = 0;
int& ref = *&x; // взять адрес а затем разыменовать
int* ptr = &x;
int* ptr2 = &*ptr // разыменовать, а затем взять адресс переменной
```

Константность указателя

На константную переменную **НЕ** может указывать обычный указатель:

```
const int x = 5;
int* ptr = &x;  // compile error
int& ref = x;   // compile error
```

- аналогично для ссылок

На константную переменную может указывать указатель на константу:

```
const int x = 5;
const int* ptr = &x;
const int& ref = x;
```

- аналогично для ссылок

Константный указатель на константную переменную:

```
const int x = 5;
const int* const ptr = &x;
```

- нельзя менять (перевешивать на другую переменную) указатель и сами данные

Нулевой указатель `nullptr`

`nullptr` - нулевой указатель, необходим для инициализации указателей, которые никуда не указывают.

Рассмотрим следующий пример:

```
double* ptr;

if (condition) {
    ptr = &value;
```

```
}

if (ptr) {
    std::cout << *ptr << std::endl; // UB because ptr has random address
}
```

Для правильной работы кода, необходимо инициализировать указатель значением `nullptr`:

```
double* ptr = nullptr;
```

Арифметика указателей

Над указателями можно проводит **арифметические операции** сложения и вычитание указателя с числом, инкремент/декремент.

Арифметика указателей приводит к смещению указателя на другой блок памяти, поскольку указатель знает тип, на адресс объекта которого указывает, то известен и размер шага.

При арифметике указателей изменяется адрес, хранимый указателем на столько же байт, сколько отведено под хранение переменной типа.

Естественно это работает для неконстантных указателей, которые могут быть перевешаны на другой объект.

Сравнение указателей

Можно производить сравнение указателей, сравнение с `nullptr`.

Указатель на начало массива будет меньше указателя на конец.

Работает, в том числе, для константных указателей.

Расстояние между указателями

Можно определить расстояние между двумя указателями:

```
int arr[5] = {1, 2, 3, 4, 5};
int* begin = arr;   // pointer to 1
int* end = arr + 5; // pointer to pos after end of array
auto distance = end - begin;  // 5 type is std::ptrdiff_t
```

- `std::ptrdiff_t` - тип используемый для расстояния между указателями

Передача массива через указатели

Возможно передать массив в функцию через пару указателей.

Синтаксис функции: `<return_type> <func_name>(<type>* begin, <type>* end)`

Получение указателей на начало и конец:

```
int arr[5] = {1, 2, 3, 4, 5};
int* begin = arr;   // pointer to 1
int* end = arr + 5; // pointer to pos after end of array
```

В заголовочном файле `<iterator>` есть удобные функции для получения указателя на начало (`std::begin()`) и конец (`std::end()`) массива. Способ более предпочтителен для современного C++

```
int arr[5] = {1, 2, 3, 4, 5};
int* begin = std::begin(arr);   // pointer to 1
int* end = std::end(arr);       // pointer to pos after end of array
```

Пример с функцией:

```
void Print(int* begin, int* end) {
    for (; begin != end; ++begin) {
        std::cout << *begin << std::endl;
    }
}

int main() {
   int arr[5] = {1, 2, 3, 4, 5};
   int* begin = std::begin(arr);   // pointer to 1
   int* end = std::end(arr);       // pointer to pos after end of array
   Print(begin, end);
   Print(std::begin(arrv), std::end(arr));
}
```

Ссылка на указатель

Можно объявить ещё одно имя для указателя посредством ссылки:

```
int x = 0;
int* ptr = &x;
int*& ptr2 = ptr;
```

Указатель на указатель

Можно получить указатель на указатель. Нужно помнить, что каждая вложенность указателя увеличивает уровень косвенности (не прямого обращения), что может замедлять работу с исходным объектом под указателем.

```
int x;
int* ptr = &x;            // pointer to int
int** pptr = &ptr;        // pointer to pointer to int
int* const* npptr = &ptr; // non-const pointer to const pointer to int
```

Поскольку в языке есть преобразование `array to pointer`, то функцию `main` с аргументами можно записать двумя способами:

- `int main(int argc, char* argv[]) {<body>}`
- `int main(int argc, char** argv) {<body>}`

Указатель на `void`

Указатель на void — это указатель общего назначения, который может хранить адрес объекта любого типа, но не имеет информации о типе этого объекта.

Синтаксис: `void* <ptr_name>;`

Особенности:

- Может указывать на данные любого типа
- **НЕ** может быть разыменован напрямую
- **НЕ** поддерживает арифметику указателей
- Требует явного приведения для использования

Оператор `reinterpret_cast`

- `reinterpret_cast` - это оператор приведения, который интерпретирует битовое представление объекта одного типа как битовое представление объекта другого типа.

Синтаксис: `reinterpret_cast<new_type>(<expression>)`

- `new_type` - тип в который преобразовать
- `<expression>` - выражение, результат которого необходимо преобразовать

Не выполняет никаких преобразований.

Позволяет считать битовое представление числа в памяти:

```
int number = 0x12345678;
int* ptr_int = &number;
```

```cpp
char* ptr_char = reinterpret_cast<char*>(ptr_int);

std::cout << "original int = " << std::hex << number << std::endl;
std::cout << "reinterpret as bytes = ";
for (int i = 0; i < sizeof(int); ++i) {
    std::cout << std::hex << static_cast<int>(char_ptr[i]) << " ";
}
```

Указатель на функцию

Указатель на функцию — это переменная, которая хранит адрес функции, а не данных.

Позволяет:

- Передавать функции как аргументы в другие функции
- Создавать массивы функций
- Реализовывать стратегии и `callback` вызов функции

Синтаксис объявление указателя на функцию: `<return_type> (*<ptr_name>)(<arg_types>);`

- `<return_type>` - возвращаемое значение функции
- `<ptr_name>` - имя указателя
- `<arg_types>` - типы аргументов функции

Преобразование *function to function pointer*

В языке присутствует стандартное преобразование функции в указатель на функцию. Поэтому там, где требуется указатель на функцию не обязательно передавать адрес функции, можно просто передать имя функции.

Сложные объявления с указателями

Ниже приведены примеры сложных объявлений с участием указателей:

```cpp
int* arr[10];                   // array with 10 int* elements
int (*ptr_arr)[10];             // pointer to array with 10 int elements
void (*ptr_func)(int)           // pointer to function with type void(int)
void (*arr_ptr_func[10])(int)   // array with 10 elements of ptr to f void(int)
void (*(*arr_ptr_func_r_ptr_func[10])(int))(int)
```

Передача аргументов в функцию

Передача по значению, копируем объект:

```
int func(std::string s);
```

Передача по ссылке, не копируем объект, можем изменять существующий:

```
int func(std::string& s);
```

Передача по константной ссылке, не копируем объект, нельзя изменять существующий:

```
int func(const std::string& s);
```

Передача по указателю, не копируем объект, можем изменять существующий:

```
int func(std::string* s);
```

Передача по указателю на константу, не копируем объект, нельзя изменять существующий:

```
int func(const std::string* s);
```

Продление жизни ссылки на константу

Синтаксис: `const <type>& ref = <entity>`

- Константная ссылка создает себе объект и становится его именем

Константные ссылки можно инициализировать через ***rvalue*** (*lifetime expansion*)

Почему нет продления жизни не константной ссылки

```
void func(size_t& counter) {
    ++counter;
}

int main() {
    int x = 0;
    func(x);
    std::cout << x << std::endl;
}
```

Висячая ссылка (*dangling reference*)

В C++ не запрещено возвращать ссылку на локальную переменную, но поскольку по выходу из функции локальная переменная уничтожается, то ссылка становится висячей (битой) и обращение к ней приведет к UB необратимым последствиям:)

```
int& func(int& value) {
    int x = 5;
    return x;
}

int main() {
    int value = 5;
    int& x = func(value);
    std::cout << x << std::endl; // UB
}
```

Компилятор может выдавать предупреждение, это зависит от версии.

Невалидный указатель (*invalid pointer*)

Нельзя разыменовывать указатель, который указывает на память, уже не принадлежащую объекту. Шучу, это же C++ разыменовать можно, но программа скорей всего упадет, так как разыменование невалидного указателя это **UB**!

```
int main() {
    int* ptr = nullptr;
    {
        int x = 42;
        ptr = &x;
    }
    std::cout << *ptr << std::endl; // UB
}
```