3.8. Понятие области видимости и времени жизни объекта. Конфликты имен переменных, замещение менее локального имени более локальным именем. Пример ситуации неоднозначности при обращении к переменной.

Непосредственное управление автоматической памятью – выделение памяти под локальные объекты при их создании и освобождение занимаемой объектами памяти при их разрушении – осуществляется компилятором. Собственно, по этой причине память и называют автоматической.
Период времени от момента создания объекта до момента его разрушения, т. е. период времени, в течение которого под объект выделена память, называют временем жизни объекта.
Время жизни локальных объектов определяется областью видимости их имён. Область видимости локального имени начинается с места его объявления и заканчивается в конце блока, в котором это имя объявлено. Область видимости аргументов функции ограничивается телом функции, т. е. считается, что имена аргументов объявлены в самом внешнем блоке функции.

Примеры:

1. Глобальная область
2. Область пространства имен
3. Локальная область
4. Область класса

```
int main() {
    int a;

    {
        int b;
        // здесь доступны переменные a и b
    }

    {
        int a;
        // более локальная переменная перебивает менее локальную
    }
}
```

Пример неоднозначности при обращении к переменной:

0.0.1 Конфликт имёт переменных. Замещение менее локального имени более локальным именем: скрытие с глобальной областью видимости

Конфликт имён - когда в области видимости одной переменной объявленая другая переменная с тем же названием.

Можно скрыть имена с глобальной областью, явно объявляя одно и то же имя в области видимости блока. Однако доступ к именам глобальных областей можно получить с помощью оператора разрешения области (::).

```
    int i = 7;   // i has global scope, outside all blocks
    int main() {
```

```
        int i = 5;   // i has block scope, hides i at global scope
        cout << "Block-scoped i has the value: " << i << "\n";
        cout << "Global-scoped i has the value: " << ::i << "\n";
    }
    //Output:
    //Block-scoped i has the value: 5
    //Global-scoped i has the value: 7
```

В терминологии C++ есть понятие unqualified-id и qualified-id. Если есть префикс (пример: std), то id первого типа, иначе – второго.

```
#include <iostream>

namespace A {
  int x = 1;
}

namespace B {
  int x = 2;
}

using namespace A;
using namespace B;


int main() {
  std::cout << x; // CE (error: reference to 'x' is ambiguous)
}
```

3.9. Указатели и допустимые операции над ними. Сходства и различия между указателями и массивами.

Указатель — переменная, значением которой является адрес ячейки памяти. Шаблон: type* p. Требует 8 байт для хранения (чаще всего).

Операции, которые поддерживает указатель:

1. Унарная звёздочка, разыменование: $T* -> T(*p)$
   Возвращает значение объекта

2. Унарный амперсанд: $T-> T*(\&p)$
   Возвращает адрес объекта в памяти

3. +=, ++, −−, -=

4. ptr + int

5. ptr - ptr, который возвращает разницу между указателями (ptrdiff_t)

Еще есть указатель на void, void* обозначает указатель на память, под которым лежит неизвестно что. Его нельзя разыменовать.

```
#include <iostream>

struct Foo {
    void bar() {
        std::cout << "bar";
    }
}
```

```
8
9  int main() {
10     int* a = new int();
11     int* b = new int();
12     Foo* foo = new Foo();
13
14     // Операции:
15     std::cout << *a; // разыменование
16     foo->bar(); // вызов метода или поля у сложного типа
17     std::cout << *(a + 1); // прибавление числа
18     std::cout << b - a; // разница между указателями (ptrfdiff_t)
19
20     void* x = nullptr; // значение по умолчанию
21
22     int* array = new int[10];
23     std::cout << array[3] == *(array + 3); // true
24
25     // Освобождение памяти - различие у указателя и массива:
26     delete a;
27     delete[] array;
28
29 }
30
31 Nullptr - ключевое слово, введенное в C++11 для описания константы нулевого указателя.
       Данная константа имеет тип std::nullptr\_t. Nullptr является константой r-value
      .\\Что
32  будет, если разыменовать nullptr? UB. Не поддерживает вывод.Грань
33
34  между массивами и указателями весьма тонкая. Массивы являются собственным типом вида
      int(*)[size]. Что можно делать с массивом:
35 \begin{enumerate}
36     \item Привести массив к указателю это( будет указатель на первый элемент)
37     \item К массиву можно обращаться по квадратным скобкам, как и к указателю. a[0] ==
       *(a+0).
38 \end{enumerate}Для
39  удаления массива необходимо использовать delete[]. Формально это другой оператор.
```

3.10. Ссылки. Объяснение концепции. Отличия от указателей. Особенности инициализации ссылок, присваивания ссылкам. Передача аргументов по ссылке и по значению. Проблема висячих ссылок, пример ее возникновения.

Мотивация: если объявление нового объекта это новое имя для старого, то компилятору необходимо было бы решать, когда удалять старый объект и хранить в runtime дополнительную информацию.

Можно считать, что ссылка - это просто переименование объекта, и код никак не различает ссылку и сам объект. (На самом деле есть способ это сделать, но не очень-то и нужно).

В отличие от указателя, ссылка не может быть пустой, она всегда должна на что-то ссылаться.

Ссылка — особый тип данных, являющийся скрытой формой указателя, который при

использовании автоматически разыменовывается. Ссылка — это новое название для уже существующей переменной.
Различия

- Нельзя объявить массив ссылок. (any kind of arrays)
- У ссылки нет адреса. (no references to references)
- no pointers to references. Примеры:

```
    // this WILL NOT compile
    int a = 0;
    int&* b = a;

    // but this WILL
    int a = 0;
    int& b = a;
    int* pb = &b; //pointer to a

    // and this WILL
    int* a = new int;
    int*& b = a; //reference to pointer - change b changes a

```

- Существует арифметика указателей, но нет арифметики ссылок.
- Ссылка не может быть изменена после инициализации.
- Указатель может иметь «невалидное» значение с которым его можно сравнить перед использованием. Если вызывающая сторона не может не передать ссылку, то указатель может иметь специальное значение nullptr (т.е. ссылка, в отличии от указателя, не может быть неинициализированной):

```
    void f(int* num, int& num2) {
       if(num != nullptr) {} // if nullptr ignored algorithm
       // can't check num2 on need to use or not
    }

```

- Ссылка не обладает квалификатором const

```
    const int v = 10;
    //int& const r = v; // WRONG!
    const int& r = v;

    enum {
        is_const = std::is_const<decltype(r)>::value
    };

    if(!is_const) \\ code will print this
        std::cout << "const int& r is not const\n";

```

Проблема висячих ссылок

```
#include <iostream>

int& bad(int x) {
  ++x;
  return x;
```

```
}


int main() {
  int y = bad(0); // время жизни объекта закончилось, а ссылки - нет
  std::cout << y; // UB
}
```

3.11. Константы, константные указатели и указатели на константу. Константные и неконстантные операции. Константные ссылки, их особенности, отличия от обычных ссылок.

Замечание: Константную ссылку можно создать от любого объекта, но неконстантную ссылку от константного объекта — нельзя.

Плюсы и минусы использования того и другого:

- ссылки лучше использовать когда нежелательно или не планируется изменение связи ссылка $\longrightarrow$ объект
- указатель лучше использовать, когда возможны следующие моменты в течении жизни ссылки:
  - ссылка не указывает ни на какой объект;
  - ссылка указаывает на разные объекты в течении своего времени жизни.

Битая ссылка — ситуация, когда используется ссылка на разрушенный (чаще всего из-за выхода из области видимости) объект. Использование такой ссылки является UB.

```
    int& foo() {
        int a = 4;
        return a;
    }

    int main() {
        int a = foo();
        // can be anything, but more likely this will cause seg fault
    }
```

Дополнение: Ссылки можно делать полями классов, причем инициализировать их можно как на месте (since C++11):

```
    struct C {
        int field = 0;
        int& field_alias = field;
    };
    // OR
    struct C {
        C(int& x) : x(x) {}
        int& x;
    };
```

ВАЖНО!!! В одном из этих мест инициализация должна быть обязательно, т.к. ссылка должна быть проинициализирована на момент создания.

```

struct Person {

    int& getAge() {
        ++requests_count;
        return age;
    }

    int getAge() const {
        ++requests_count; // можем менять, так как поле помечено mutable
        return age;
    }

private:

    int age;
    mutable requests_count = 0;
}

int main() {
    const int* a = new int(); // константный указатель, нельзя менять объект под
    указателем
    int* const b = new int(); // нельзя менять сам указатель то( есть запрещены
    операции такие как инкремент, оператор присваивания и т д)

    Person p1;
    const Person p2;
    int& c = p1.getAge();
    ++a; // можем менять

    int d = p2.getAge(); // вызовется константная версия метода

}
```

3.12. Неявное приведение типов. Явное приведение типов с помощью оператора static_cast.

0.1 Приведение типов

0.1.1 Static cast

Создание новой сущности из старой. Работает на этапе компиляции. Берёт объект старого типа и возвращает нового:

```
    int main() {
        int x = 0;
        double d = static_cast<double>(x);
    }
```

Работает в том числе и с пользовательскими правилами приведения типа.

```

int foo(int x) {
    return x + 1;
}

int main() {
    double d = 3.0;
    foo(d); // здесь произошло неявное приведение типов
```

```
9       int y = static_cast<double>(d); // явно попросили компилятор привести типы
10  }
```

static_cast и неявное преобразование работает также и с пользовательскими типами, нужно только определить правила преобразования типов.