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:

```
#include <iostream>

namespace A {
  int x = 1;
}

namespace B {
  int x = 2;
}

using namespace A;
using namespace B;

```

```cpp

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.

```cpp

#include <iostream>

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

int main() {
    int* a = new int();
    int* b = new int();
    Foo* foo = new Foo();

    // Операции:
    std::cout << *a; // разыменование
    foo->bar(); // вызов метода или поля у сложного типа
    std::cout << *(a + 1); // прибавление числа
    std::cout << b - a; // разница между указателями ($ptrfdiff_t$)

    void* x = nullptr; // значение по умолчанию

    int* array = new int[10];
    std::cout << array[3] == *(array + 3); // true

    // Освобождение памяти - различие у указателя и массива:
    delete a;
    delete[] array;

}
```

Грань между массивами и указателями весьма тонкая. Массивы являются собственным типом вида int(*)[size]. Что можно делать с массивом:

1. Привести массив к указателю (это будет указатель на первый элемент)
2. К массиву можно обращаться по квадратным скобкам, как и к указателю. a[0] == *(a+0).

Для 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.
Отличия указателя от ссылки:

- 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.

Определение. *Константы* - такой тип, к которому применимы константные операции.

```
    const int x = 3;
```

Причем константы необходимо инициализировать сразу при объявлении.
Возможна передача неконстантной версии, куда необходима константая.

Определение. *Константый указатель* - это указатель, значение которого не может быть изменено после инициализации. Для объявления константного указателя используется ключевое слово const между звёздочкой и именем указателя:

```
    int value = 11;
    int* const ptr = &value;
```

Подобно обычным константным переменным, константный указатель должен быть инициализирован значением при объявлении. Это означает, что он всегда будет указывать на один и тот же адрес. В вышеприведенном примере *ptr* всегда будет указывать на адрес value (до тех пор, пока указатель не выйдет из области видимости и не уничтожится):

```
    int val1 = 14;
    int val2 = 88;
    int* const ptr = &val1;//ок, инициализация адресом val 1
    ptr = &val2;// - не ок, после инициализации нельзя менять константый указатель
```

Однако, поскольку переменная *val*, на которую указывает указатель, не является константой, то её значение можно изменить путем разыменования константного указателя:

```
    int val = 11;
    int* const ptr = &val;
    *ptr = 8;//ок, так как тип данных (int) - не константый
```

Определение. *Указатель на константное значение* — это неконстантный указатель, который указывает на неизменное значение. Для объявления указателя на константное значение, используется *ключевое слово const перед типом данных:*

```
    const int val = 11;
    const int* ptr = &value;//ок, ptr - неконстантый указатель на const int
    *ptr = 8;//нельзя, т.к. мы не можем изменить константое значение
```

Можно также инициализировать неконстантым значением:

```
    int val = 11;
    const int* ptr = &value;
```

Указатель на константную переменную может указывать и на неконстантную переменную (как в случае с переменной *val* в примере, приведенном выше). Подумайте об этом так: указатель на константную переменную обрабатывает переменную как константу при получении доступа к ней независимо от того, была ли эта переменная изначально определена как *const* или нет. Таким образом, следующее в порядке вещей:

```
    int val = 11;
    const int* ptr = &val;//ptr указывает на const int
    val = 8;//все ок, val - не константа
```

Но не следующее:

```
    int val = 11;
    const int *ptr = &val; // ptr указывает на "const int"
    *ptr = 12; // ptr обрабатывает value как константу, поэтому изменение значения
    //переменной val через ptr не допускается
```

Указателю на константное значение, который сам при этом не является константным (он просто указывает на константное значение), можно присвоить и другое значение:

```
    int val1 = 11;
    const int *ptr = &val1; // ptr указывает на const int
    int val2 = 12;
    ptr = &val2; // хорошо, ptr теперь указывает на другой const int
```

Наконец, можно объявить константный указатель на константное значение, используя *ключевое слово const как перед типом данных, так и перед именем указателя:*

```
    int val = 11;
    const int *const ptr = &val;
```

Константный указатель на константное значение нельзя перенаправить указывать на другое значение также, как и значение, на которое он указывает, — нельзя изменить.

Определение. *Константное выражение* - это то, которое можно посчитать в compile time.

Более подробно о полной классификации можно прочитать здесь.

Определение. *Константная ссылка.* Объявить ссылку на константное значение можно путем добавления ключевого слова const перед типом данных:

```
    const int value = 7;
    const int &ref = value; // ref - это ссылка на константную переменную value
```

Ссылки на константные значения часто называют просто *«ссылки на константы»* или *«константные ссылки»*.

В отличие от ссылок на неконстантные значения, которые могут быть инициализированы только неконстантными l-values, ссылки на константные значения *могут быть инициализированы неконстантными l-values, константными l-values и r-values*:

```
    int a = 7;
    const int &ref1 = a; // ок: a - это неконстантное l-value
    const int b = 9;
    const int &ref2 = b; // ок: b - это константное l-value
    const int &ref3 = 5; // ок: 5 - это r-value
```

Как и в случае с указателями, константные ссылки также могут ссылаться и на неконстантные переменные. При доступе к значению через константную ссылку, это значение автоматически считается const, даже если исходная переменная таковой не является:

```
    int value = 7;
    const int &ref = value; // создаем константную ссылку на переменную value
    value = 8; // ок: value - это не константа
    ref = 9; // нельзя: ref - это константа
```

Когда константная ссылка инициализируется значением r-value, время жизни r-value продлевается в соответствии со временем жизни ссылки:

```
   int somefcn()
    {
        const int &ref = 3 + 4;
        /*обычно результат 3 + 4 имеет область видимости выражения
        и уничтожился бы в конце этого стейтмента, но, поскольку
        результат выражения сейчас привязан к ссылке на константное значение,*/
        std::cout << ref; // мы можем использовать его здесь
    } // и время жизни r-value продлевается до этой точки,
    //когда константная ссылка уничтожается
```

Ссылки на константные значения особенно полезны в качестве параметров функции из-за их универсальности. Константная ссылка в качестве параметра позволяет передавать неконстантный аргумент l-value, константный аргумент l-value, литерал или результат выражения:

```
    #include <iostream>
    void printIt(const int &a)
    {
        std::cout << a;
    }

    int main()
    {
        int x = 3;
        printIt(x); // неконстантное l-value
```

```
        const int y = 4;
        printIt(y); // константное l-value
        printIt(5); // литерал в качестве r-value
        printIt(3+y); // выражение в качестве r-value
        return 0;
}
```

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

Определение. *Неявное преобразование типов*, когда компилятор автоматически конвертирует один фундаментальный тип данных в другой.

Неявное преобразование типов (или «автоматическое преобразование типов») выполняется всякий раз, когда требуется один фундаментальный тип данных, но предоставляется другой, и пользователь не указывает компилятору, как выполнить конвертацию (не использует явное преобразование типов через операторы явного преобразования).

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

int main() {
    double d = 3.0;
    foo(d); // здесь произошло неявное приведение типов
    int y = static_cast<int>(d); // явно попросили компилятор привести типы
}
```

static_cast - создание новой сущности из старой. Работает на этапе компиляции. Берёт объект старого типа и возвращает нового.

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