Московский физико-технический институт
Физтех-школа прикладной математики и информатики

П Р О Г Р А М М И Р О В А Н И Е Н А С + +

I СЕМЕСТР

Лектор: *Мещерин Илья*

Авторы: *Дамир Ачох*
Проект на Github

осень 2021

Содержание

1 3-я лекция

1.1 Объявления, определения и области видимости

Определение 1.1. Объявление (declaration) - введение новой сущности (переменной, функции и т.д.)

Пример. `int x = 5;`

```
int main() {

}
```

Замечание. Нельзя объявлять функции внутри других функций.

Определение 1.2. Определение (definition) - задать поведение или значение сущности.

Замечание. Для переменной в большинстве случаев объявление переменной является также ее определением.

Пример. Пример объявления без определения:

```
int g(int x);
```

Замечание. Любое определение является объявлением, но не наоборот.

Замечание. Объявлять можно сколько угодно раз. Определять - только один.

```
int x = 3;

int main() {
    int x = 5;
}
```

Это разные переменные x. При этом локальная «затмевает» глобальную. Можно создавать новый scope (область видимости) - просто написать {}.

```
int x = 3;
int main() {
    int x = 5;
    {
        int x = 10;
    }
}
```

В одной области видимости не может быть переменных с одним именем. По имени берется переменная лежащая в ближайшем области видимости.

Чтобы обратиться к глобальной переменной из любой области видимости нужно написать ::x.

```
int x = 3;
int main() {
    int x = 5;
    cout << ::x << '\n'; \\ 3
}
```

Обратиться к локальной переменной из другой области видимости, имеющее то же самое название, что и переменная из текущей области видимости невозможно.

1.2 Namespace

Способ разграничить области видимости.

```
namespace N {
    int x = 5;

    int f(int x);

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

Namespace нельзя открывать внутри других сущностей, кроме других namespace. Чтобы обратиться к переменной namespace'а, нужно написать N::x.

Namespace можно открывать и закрывать сколько угодно раз. При повторном открытии namespace'а происходит добавление сущностей в него. Вне namespace'а можно определять его функции (если они не были определены раньше). При таком определении нужно обязательно объявить сущность внутри самого namespace'а.

Qualified-id - явно указано, откуда взята сущность (N::x).

```
using vi = std::vector<int>;
```

Грубо говоря, сокращение std::vector<int>. Но этот способ отличается от define'ов (define - потекстовая замена, как Ctrl+R). typedef - C-style способ.

```
using N::x;
```

Этой командой переменная x из namespace'а N переносится в текущую область видимости.

```
int main() {
    using N::x;
    int x = 5;
}
```

Это ошибка!

```
using namespace N;
```

Перекидываются все сущности namespace'а. Плохая практика!

1.3 Выражения и операторы

```
a + 5; (--a * b << 3) + 1;
```

Это выражения (expressions). Их можно писать только внутри функций! Нельзя в глобальной области, в namespace и т.д.

Операторы:

▷ Арифметические

▷ Побитовые

▷ Сравнения (since C++20, <=>)

▷ Логические

▷ Инкремент и декремент ($++x$ - префиксный, $x++$ - постфиксный)

▷ Присваивания

Инкремент и декремент - единственные операторы, меняющие аргумент. Префиксный инкремент: возвращает уже увеличенное значение переменной (при этом возвращаемый тип - ссылка). Постфиксный возвращает прошлое значение переменной (только значение, не ссылка), но все равно увеличивает переменную.

Пример. `x++ && ++x = false, при x = 0 (после этой команды, x = 1)`

Присваивание возвращает ссылку на изменённую переменную.

Замечание. `x += 5;`
`x = x + 5;`

Это на самом деле две разные команды. Первая сразу увеличивает переменную на 5. Вторая считает временное значение x + 5, потом присваивает это значение в x. Начиная на с C++17 гарантируется, что правая часть присваивания считается раньше левой. Например, ++x = x++. После этой команды x не изменится.

Присваивания выполняются справа-налево (x = y = 5, x = y += 5).

2 4-я лекция

2.1 Понятия l-value, r-value

Сейчас введём их неформально: l-value - то, что может стоять слева от оператора присваивания, r-value - то, что не может.

```
int x = 1; \\ OK
x + 5 = 10; \\ Wrong
10 = x; \\ Wrong
++x = 5; \\ OK
x++ = 5; \\ Wrong
(a = b) = c; \\ OK
(x + 5)++; \\ Wrong
(x = 5)++; \\ OK
(x = x + 1) = 10; \\ OK
```

Замечание. Знак равно не всегда оператор. Например:

```
int x = 5;
```

Тут = - не оператор, а просто символ.

2.2 Тернарный оператор

```
a ? b : c
```

Если a - true, то вычисляется и возвращается b (c даже не вычисляется). Иначе вычисляется и возвращается c (b даже не вычисляется).

Тернарный оператор возвращает l-value, тогда и только тогда, когда и b и c - l-value.

```
(a == 1 ? x++ : ++x) = 10; // Wrong
```

Это будет CE, независимо от того, чему равно a, т. к. ++x - r-value.

2.3 Оператор «запятая»

```
x = 1, y = 3, ++z;
```

Оператор «запятая» сначала высчитывает левый операнд, потом высчитывает правый операнд и возвращает то, что вернул правый операнд.

```
(x = 5, ++x) = 1; \\ OK
```

Однако «запятая» не всегда оператор.

```
int x = 2, y = 1; \\ Not operator
f(x, y); \\ Not operator
```

Приоритет оператора «запятая» самый маленький из всех.

```
int x = 1, 5; \\ x = 1, but return value - 5;
```

2.4 Subscription (обращение по индексу)

```
a[5]; \\ l-value
```

Оператор «квадратные скобки» - обращение по индексу (возвращает l-value).

2.5 Sizeof

Sizeof(x) - это оператор, возвращающий кол-во байт, которое занимает переменная x. При этом компилятор не вычисляет само значение x.

```
cout << sizeof(++x); \\ 4
```

От этой команды x не поменяется. Sizeof() вычислят кол-во байт, которое занимала бы переменная, будь она посчитанная.

2.6 Приоритет операторов

Таблица приоритетов

2.7 Ошибка компиляции (CE)

Бывает трёх видов: лексические, синтаксические и семантические.

3 5-я лекция

3.1 Runtime Error (RE)

```
int a = 5 / 0;
```

Целочисленное деление на ноль - это RE.

Segmentation fault (Segfault) - ошибка сегментирования (обращение к элементу массива, которого не было, и не только) - тоже RE. Это ошибка на уровне системы, не на уровне языка (также как Floating point exception).

3.2 Undefined Behaviour

▷ Обращение за границу массива (segfault - частный случай UB)

▷ Переполнение int (классный пример представлен в лекции)

3.3 Unspecified Behaviour

Подробнее в лекции

```
int f() {
    cout << 1;
    return 1;
}

int g() {
    cout << 2;
    return 2;
}

int h() {
    cout << 3;
    return 3;
}

int main() {
    cout << f() * g() + h(); \\ ***5
}
```

В конце точно выведется 5, а что будет до этого - неизвестно (f, g, h могут считаться в каком угодно порядке). Порядок вычислений не то же самое, что приоритет операторов.

3.4 Указатели

У каждой переменной есть свой адрес (&a - получить адрес переменной a). Возвращаемый тип этой операции - T^*, где T - тип переменной. Операнд к & должен быть l-value. Указатель - переменная, хранящая адрес другой переменной (тип T^*). Можно взять адрес указателя, т. к. указатель - тоже переменная.

```
int** p = &(&a); \\ Wrong (&a - r-value)
```

Обратная операция - * (по адресу вернуть значение, хранящееся по этому адресу). *: $T^* \to T$.

```
int* a, b;
```

В итоге, a - это указатель, b - нет. Поэтому не рекомендуется объявлять несколько переменных на одной строке.

```
int a = 1;
int* p = &a;
{
    int b = 2;
    p = &b;
}
cout << *p; \\ UB
```

3.5 Реализация swap

```
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

3.6 Операции над указателями

▷ Инкремент - перейти в следующую ячейку памяти, сдвинутую на sizeof(T) байт

▷ Декремент

▷ Сложение с int - сделать инкремент n раз

▷ Разность указателей - сколько между ними шагов размера sizeof(T).

4 6-я лекция

4.1 NULL vs nullptr

Пустой указатель, ни на что не указывает. NULL - с языка C. nullptr - рекомендованный способ.

4.2 Виды памяти