

версия от 17 сентября 2023 г.

**С. В. Ветров**

Черновик учебного пособия

# **Объектно-ориентированное программирование на C++**

# Оглавление

<b>Введение</b>	<b>4</b>
<b>1 Введение в C++</b>	<b>5</b>
1.1 Характеристика языка . . . . .	5
1.2 Основы . . . . .	8
1.2.1 Основные типы данных и операции с ними . . . . .	9
1.2.2 Константы и выражения времени компиляции . . . . .	15
1.2.3 Перечисления (enum) . . . . .	16
1.2.4 auto, typedef . . . . .	17
1.2.5 Преобразования типов . . . . .	18
1.2.6 Приоритет операторов . . . . .	19
1.2.7 Ввод и вывод на экран . . . . .	19
1.2.8 Операторы . . . . .	21
1.2.9 Оператор присваивания . . . . .	21
1.2.10 Логические операции . . . . .	22
1.2.11 Тернарный условный оператор . . . . .	22
1.2.12 Управляющие операторы . . . . .	23
1.3 Динамическая память, указатели и ссылки . . . . .	25
1.3.1 Указатели . . . . .	25
1.3.2 new и delete . . . . .	27
1.3.3 Ссылки . . . . .	27
1.3.4 Спецификатор const и указатели . . . . .	28
1.3.5 Проблемы динамической памяти . . . . .	28
1.4 Функции . . . . .	29
1.4.1 Перегрузка функций . . . . .	33
1.4.2 Значения параметров по умолчанию . . . . .	34
1.4.3 inline-функции . . . . .	34
1.4.4 static функции и локальные переменные . . . . .	34
1.4.5 Спецификатор constexpr . . . . .	34
1.4.6 Выводы и рекомендации . . . . .	34
1.5 Пространства имён (namespaces) . . . . .	35
1.6 Массивы . . . . .	38
1.6.1 Статические массивы . . . . .	38

1.6.2	Массив из символов . . . . .	40
1.6.3	Функции для работы с массивами . . . . .	40
1.6.4	Динамические массивы . . . . .	41
1.7	Устройство памяти программы . . . . .	45
1.8	Заголовочные файлы и сrr файлы . . . . .	46
1.9	Модули C++20 . . . . .	49
1.10	Компиляция программы . . . . .	50
1.10.1	Компиляция программы из одного файла . . . . .	50
1.10.2	Макросы препроцессора . . . . .	50
1.10.3	Этапы компиляции . . . . .	51
1.10.4	Компиляция нескольких файлов и статической библиотеки	51
1.10.5	Параметры компиляции, настройки компиляции в IDE .	52
1.11	Параметры командной строки . . . . .	53
<b>2</b>	<b>Продвинутые возможности языка</b>	<b>54</b>
2.1	Обработка исключительных ситуаций . . . . .	54
2.1.1	Условный оператор, try ... catch, assert . . . . .	59
2.2	Функции . . . . .	60
2.2.1	Статические локальные переменные . . . . .	60
2.2.2	Переменное число параметров функции (variadic arguments)	60
2.2.3	Указатель на функцию . . . . .	61
2.2.4	Перегрузка операторов (operator overloading) . . . . .	62
2.2.5	Анонимные функции . . . . .	63
2.2.6	Автоматическое тестирование . . . . .	65
2.3	Шаблонные функции . . . . .	70
2.4	Ассемблерные вставки . . . . .	72
2.5	Инструменты разработчика . . . . .	73
2.5.1	Обзор . . . . .	73
2.5.2	Visual Studio . . . . .	74
<b>3</b>	<b>Стандартная библиотека</b>	<b>76</b>
3.1	string . . . . .	76
3.2	string stream . . . . .	77
3.3	Умные указатели . . . . .	77
3.4	Контейнеры . . . . .	79
3.4.1	Общее описание . . . . .	79
3.4.2	vector . . . . .	80
3.5	Алгоритмы . . . . .	81
3.6	Файловые потоки . . . . .	82
3.7	Регулярные выражения (regex) . . . . .	84
3.8	Взаимодействие с ОС . . . . .	86

3.9	Время . . . . .	86
<b>4</b>	<b>Введение в объектно-ориентированное программирование</b>	<b>88</b>
4.1	Предпосылки появления ООП . . . . .	88
4.2	Абстрактный тип данных . . . . .	88
4.3	Классы в C++ . . . . .	92
4.3.1	Конструкторы и деструктор . . . . .	98
4.3.2	Работа с экземплярами класса . . . . .	104
4.3.3	Статические члены класса (static members) . . . . .	108
4.3.4	Дружественные функции и классы . . . . .	110
4.3.5	Перегрузка операторов . . . . .	110
4.3.6	Представление класса в UML . . . . .	111
4.4	Шаблонные классы . . . . .	112
4.5	Модульное тестирование . . . . .	113
<b>5</b>	<b>Отношения между классами</b>	<b>116</b>
5.1	Ассоциация . . . . .	116
5.2	Агрегация и композиция . . . . .	116
5.3	Наследование (inheritance) . . . . .	119
5.3.1	Абстрактные классы . . . . .	125
5.3.2	Когда использовать наследование . . . . .	127
5.4	Динамический полиморфизм . . . . .	128
<b>6</b>	<b>SOLID</b>	<b>134</b>
6.1	Принцип единственной ответственности . . . . .	134
6.2	Принцип открытости и закрытости . . . . .	134
6.3	Принцип подстановки . . . . .	134
6.4	Принцип разделения интерфейсов . . . . .	134
6.5	Принцип инверсии зависимостей . . . . .	134
<b>7</b>	<b>Шаблоны проектирования</b>	<b>135</b>
7.0.1	Обзор . . . . .	135
7.0.2	Некоторые примеры . . . . .	135
	<b>Заключение</b>	<b>136</b>
	<b>Библиографический список</b>	<b>137</b>
	<b>Предметный указатель</b>	<b>138</b>

# Введение

Пособие освещает только основы языка C++ и ООП.

Предполагается, что читатель уже знаком с одним из императивных языков программирования.

Примеры кода на языке C++ приведены для стандарта C++19, проверены компилятором G++11.2.

Материалы дисциплины: [github.com/VetrovSV/OOP](https://github.com/VetrovSV/OOP)

# 1 Введение в C++

*Си позволяет легко выстрелить себе в ногу; с C++ это сделать сложнее, но, когда вы это делаете, вы отстреливаете себе ногу целиком.*

*Ограничение возможностей языка с целью предотвращения программистских ошибок в лучшем случае опасно.*

---

## 1.1 Характеристика языка

- Построен на основе C
- Общего назначения
- Компилируемый
- Статическая типизация
- Слабая типизация
- Объектно-ориентированный
- Ручное управление памятью (без сборщика мусора)

**Статическая типизация** (static type checking) – вид типизации, при которой переменная, аргумент функции или возвращаемое значение связываются с типов *во время компиляции* (compiler time) . Противоположный подход к типизации – **динамическая типизация** (dynamic type checking), при которой тип переменной, аргумента или возвращаемого значения могут изменяться во время работы программы (run-time).

```
// C++
int i = 5;
i = "кря-кря"; // синтаксическая ошибка
```

```
// Python, динамическая типизация
i = 5;           // тип определяется из значения: int
i = "кря-кря";  // тип переменной изменён: str
```

**Слабая типизация** (weak typing) – типизация, при которой возможно неявное преобразование типов, даже если возможна потеря точности. Такую типизацию ещё называют не строгой. Она противопоставляется **сильной типизации** (строгой типизации), которая запрещает смешивать в выражениях разные типы, без явных вызовов преобразования. Понятия сильной и слабой типизации не имеют четкого определения и чаще используются для сравнения системы типизации в языках. Сильная типизация есть в языках программирования: Java, Haskell, Python. Слабая: C, C++, JavaScript.

```
// Java
int i = 5 + true;
// error: bad operand types for binary operator '+'

// C++
int i = 5 + true;
// Код синтаксически верен
```

Сборщик мусора (garbage collector) – специальный процесс, работающий параллельно с программой, который освобождает выделенную программе, но более не используемую память. Сборщики мусора типичны для интерпретируемых (Python) и компилируемых в байт-код (C#, Java) языков программирования.

Неформальная характеристика языка:

- Большое сообщество программистов, большая коллекция библиотек, справочной информации.
- Популярен в течении последних 30+ лет, развитая стандартная библиотека.
- Активно развивается: новые стандарты выходят каждые 2-3 года: C++98, C++03, C++11, C++14, C++17, C++19, C++20, C++23.
- Множество реализаций для всех популярных ОС.
- Поддерживает многие концепции программирования (ООП, динамическое управление памятью, анонимные функции, шаблоны ...) которые есть в других языках программирования.
- Особенно широко используется там, где высоки требования к производительности.

[en.cppreference.com/w/cpp/language/history](https://en.cppreference.com/w/cpp/language/history) – краткое описание стандартов

## **Философия языка**

- Максимально возможная совместимость с C
- Поддержка многих парадигм программирования: процедурное, ООП, обобщенное, ...
- Не плати за то, что не используешь
- Максимально возможная независимость от платформ

## **Компиляторы и среды разработки**

Компиляторы:

- G++ ( MinGW-64 для Windows)
- MSVC (входит в состав VisualStudio)
- Clang
- Intel C++ Compiler
- ...

Среды разработки

- Visual Studio
- Qt Creator
- CLion
- ...



## 1.2 Основы

### Алфавит

...

### Структура программы

Короткий вариант главной функции программы:

```
// главная функция программы
int main(){

    // ...основной алгоритм...

    return 0;
}
```

о параметрах командной строки и возвращаемом значении функции main см. раздел 1.11

В дальнейших примерах для краткости функция main будет опускаться. Подразумевается, что все объявления констант и переменных и операторы будут приведены внутри этой функции. Включение заголовочных файлов и директивы using будут всегда приводиться до объявления функции main.

Типичная структура программы:

```
/// Краткое описание программы и указание автора

// Включение заголовочных файлов стандартной библиотеки. Например:
#include <iosream>
// Включение заголовочных файлов (собственных и из сторонних библиотек)
#include "my_unit.h"           // пояснение, если необходимо

// Включение содержимого пространств имён в текущую область видимости. Например:
using namespace std;

int main(){
    // ...основной алгоритм...
}
```

рекомендуется отделять разные по смыслу и назначения участки кода одной и двумя пустыми строками

### Компиляция программы

Скомпилируем программу компилятором C++ из набора компиляторов GCC (версия для Windows называется MinGW):

```
g++ main.cpp -o hello_world.exe
```

предполагается, что компилятор установлен и имя основного файла компилятора доступно из командной строки, файла main.cpp находится в той же папке, откуда вызвана команда

После ключа `-o` указывается имя исполняемого файла. Более подробно процесс компиляции программы рассматривается в разделе 1.10.

### 1.2.1 Основные типы данных и операции с ними

**Объявление (*declaration*)** включает в себя указание идентификатора (имени), типа, а также других аспектов элементов языка, например, переменных и функций. Объявление используется, чтобы уведомить компилятор о существовании элемента.

**Определение (*definition*)** включает в себя объявление, дополняя его значением (для переменной или константы) или телом функции или метода.

В C++, в отличие от Паскаля, нет специального раздела программы для определения или объявления переменных. Синтаксис объявления переменной<sup>1</sup>:

```
attr decl_specifier_seq init_declarator_list;
```

- `attr` – набор атрибутов
- `decl_specifier_seq` – спецификаторы типа
- `init_declarator_list` – набор идентификаторов с необязательной инициализацией.

**Инициализация** – задание начального значения.

Примеры объявления переменных и констант:

```
int n;           // объявление переменной
int A = 42;      // определение переменной (объявление + инициализация)
// определение константы:
const float x = 1.68;
```

Стандарт языка не устанавливает требований к значениям по умолчанию для не инициализированных переменных. Такие ситуации называются неопределённым поведением (*undefined behavior*). На практике, их значения могут быть какими угодно.

Подробнее о простых типах: [ru.cppreference.com/w/cpp/language/types](http://ru.cppreference.com/w/cpp/language/types)

---

<sup>1</sup>[en.cppreference.com/w/cpp/language/declarations](http://en.cppreference.com/w/cpp/language/declarations)

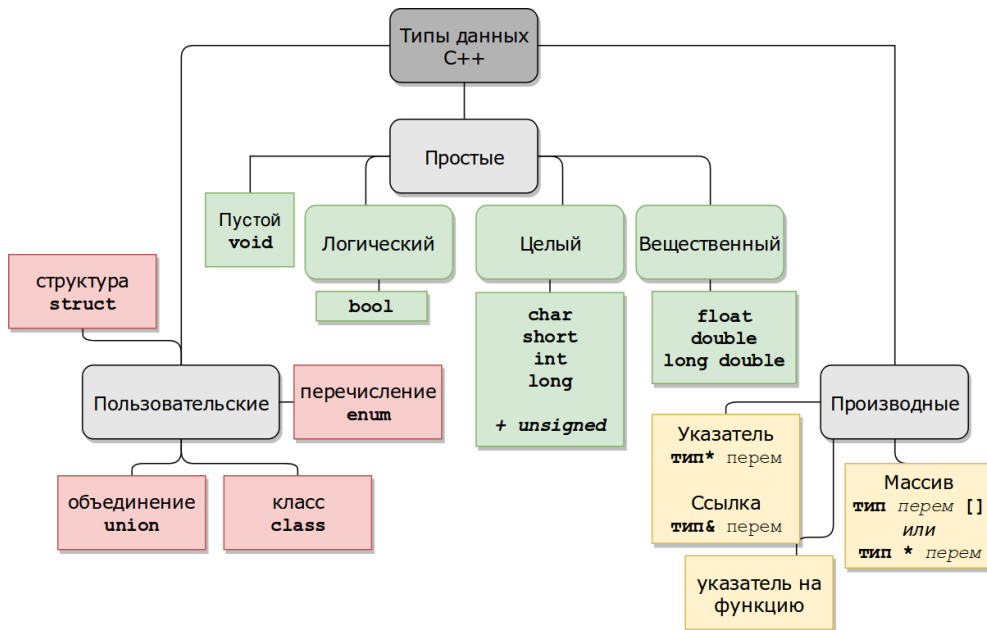


Рис. 1.1. Типы данных

## Область видимости

Переменные доступны в своей области видимости, определяемой операторными скобками или вложенностью в операторы.

```
// область видимости переменной x: функция main
// время жизни: до конца выполнения функции main
int main() {
    int x = 42;
}
```

Дополнительные операторные скобки ограничивают время жизни область видимости:

```
// область видимости переменной x: функция main
// время жизни: до конца выполнения функции main
int main() {
    int y = 0;
    {
        int x = 42;
        y = 123;
    }
    // x = 123; -- ошибка, переменная x не объявлена
}
```

```
for (int i = 0; i < 10; ++i) {
    // переменная i существует только в теле цикла
    cout << i;
}
// ошибка: идентификатор i не объявлен
cout << i;
```

Data Type	Size (bytes)	Size (bits)	Value Range
unsigned char	1	8	0 to 255
signed char	1	8	-128 to 127
char	1	8	either
unsigned short	2	16	0 to 65,535
short	2	16	-32,768 to 32,767
unsigned int	4	32	0 to 4,294,967,295
int	4	32	-2,147,483,648 to 2,147,483,647
unsigned long	8	64	0 to 18,446,744,073,709,551,616
long	8	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	64	0 to 18,446,744,073,709,551,616
long long	8	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	32	3.4E +/- 38 (7 digits)
double	8	64	1.7E +/- 308 (15 digits)
long double	8	64	1.7E +/- 308 (15 digits)
bool	1	8	false or true

Рис. 1.2. Основные числовые типы данных [компилятор ??? для 64-разрядной ОС]

## Числовые типы данных

Размер памяти, занимаемой, типами long, double, long double и другими многобайтовыми типами данных может отличаться в зависимости от разрядности (32 или 64 бита) платформы, для которой компилируется программа. Размер значения типа **bool** – 1 байт, хотя, фактически, для такой переменной достаточно одного бита. Но ОС может адресовать память по байтам, а не по битам.

[en.cppreference.com/w/cpp/language/sizeof](http://en.cppreference.com/w/cpp/language/sizeof)

```
int n; // можно не задавать значение (объявление)
float x = -47.039; // можно задавать (определение)
float y,z;
const short N = 24; // константе задавать значение обязательно
char c = 'q'; // символ
char str1[] = "qwerty"; // строка (как массив символов)
// string - тоже строка, но лучше (удобнее в работе)
string str2 = "qwerty"; // нужно подключить модуль string
n = N;
N = n; // Ошибка! Константу поменять нельзя
a = 42; // Ошибка! Переменная a не объявлена
```

## Литералы

**Литерал** (literal) или безымянная константа – запись в исходном коде, представляющая собой фиксированное значение. Литералами также называют представление значения некоторого типа данных. Например в примере выше **"qwerty"** – строковый литерал.

## Целочисленные литералы:

```
// в десятичной системе счисления:
int d = 42;
// в других системах счисления:
int o = 052;           // восьмеричная, число должно начинаться с 0
int x = 0x2a;          // шестнадцатеричная
int X = 0X2A;          // шестнадцатеричная
int b = 0b101010;      // двоичная

// для лучшей читаемости допустимо разделять разряды знаком '
long l2 = 18'446'744;
```

Больше информации о целочисленных литералах, в том числе суффиксах: [en.cppreference.com/w/cpp/language/integer\\_literal](http://en.cppreference.com/w/cpp/language/integer_literal).

В шестнадцатеричной записи для кодирования одной цифры достаточно ровно 4 бит. Соответственно любое двузначное число можно сохранить в один байт.

**Вещественные литералы** обязательно содержат точку в своей записи.

```
// веществ. литерал 1.0
float x1 = 1.0;
// дробную часть можно опускать, если вместо неё можно подставить ноль
float x2 = 1.;
float x3 = 0.7;
// аналогично можно пропускать целую часть числа, если она нулевая
float x4 = .7;

// 1 -- целочисленный литерал, неявно преобразовываемый к типу float
// при инициализации переменной
float y = 7;
```

**Экспоненциальная запись** числа  $x = 123.456 \cdot 10^{-67}$  в программе может быть представлена как

```
double x = 123.456e-67;
```

todo: Мантисса + экспонента + порядок

todo: **Комплексные числа**

Для явного указания типа литерала используют суффиксы, например:

```
auto a = 20u;           // unsigned
auto b = 20llu;         // unsigned long long
auto x = 3.40f;         // float
auto y = 3.40l;         // long double
auto z = 6.67e-11l;     // long double
```

Ключевому слову **auto**, которое заставляет компилятор вывести тип переменной по её значению, посвящён раздел 1.2.4

## Числовые операторы

% – целочисленный оператор для взятия остатка от деления.

Отдельного оператора для возведения числа в степень в C++ нет.

Инкремент (++) и декремент (--) – унарные целочисленные операторы, увеличение и уменьшение значения переменной на единицу соответственно.

```
int x = 0;
// постфиксный инкремент
cout << x++;    // -> 0; x = 1;

// префиксный инкремент
cout << ++x;    // -> 2; x = 2;
```

Документация: [en.cppreference.com/w/cpp/language/operator\\_incredec](http://en.cppreference.com/w/cpp/language/operator_incredec)

...

## Символьный тип и строковые литералы

**char** – символьный тип, занимает один байт. Можно записать либо число (код символа) либо сам символ, но при выводе (например cout) всегда будет показан символ. Символьные литералы приводятся в одинарных кавычках:

```
char = '!';
cout << 'A';
```

ASCII – American standard code for information interchange) – таблица, в которой некоторым распространённым печатным и непечатным символам сопоставлены числовые коды (рис 1.3).

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	\$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Рис. 1.3. первая половина ASCII. Dec – код символа (в десятичной система счисления)

[en.cppreference.com/w/cpp/language/ascii](http://en.cppreference.com/w/cpp/language/ascii).

ASCII ограничена 256 вариантами символов. Вторая половина (коды с 128 и далее) может использоваться для хранения кодов символов основного языка ОС (если он отличается от английского), например русского языка. см. кодировки Windows-1251, CP866 и др.

```
// коды символов латинского алфавита
char c = 65;           // 'A'
c = c + 1;
cout << c;            // 'B'
```

Для хранения символов из таблицы Unicode, которая содержит знаки почти всех алфавитов и многие другие символы восьмитонного типа char недостаточно. Поэтому рекомендуется сохранять не ASCII символы не в символьной переменной, а в массиве (**строке**), где один символ может быть закодирован несколькими байтами. Символ можно написать непосред-

ственно или привести его код в формате "\uX", где X – шестнадцатеричное число. I ♥ C++:

```
// 7 символов, 9 байт + 1 байт для обозначения конца строки
char[] str2 = "I \u2665 C++";
// 2 байта + 1 байт для обозначения конца строки
cout << sizeof("\u2665");
```

напомним, что для кодирования любой шестнадцатеричной цифры необходимо 4 бита

В конец любого строкового литерала всегда добавляется один нулевой байт, который служит признаком конца строки.

см. также раздел 1.6.2 Массив из символов.

**Экранирование символов** – замена в тексте управляющих символов на соответствующие текстовые подстановки. В C++ для экранирования используется слеш – \.

```
char str1[] = "a \\ b"; // результат: a / b
char str2[] = "Don\'t Tread On Me"; // результат: Don't Tread On Me
char str3[] = "tab\tseparated\tvalues"; // tab separated values
```

## 1.2.2 Константы и выражения времени компиляции

При объявление константы спецификатор типа **const** может быть указано слева или справа от самого типа:

```
// эти определения констант равнозначны
const int A = 42;
int const N = 42;
```

см. также раздел 1.3.4  
Спецификатор const и указатели

*todo: именование констант*

**Магические числа** – числовые литералы (значения) приведённые в исходном коде программы смысл которых не очевиден. Стоит заменять такие значения на константы, так как имя константы может говорить о смысле значения. А если такие значения повторяются, то при изменении будет достаточно изменить значение в одном месте.

Плохая практика:

```
int numbers[1024];

for (unsigned i = 0; i < 1024; i++)
    numbers[i] = rand();
```



Хорошая практика: константа вместо магического числа, имя с большой буквы, конвенциональное имя для размера массива

```
const unsigned N = 1024;    // размер массива
int numbers[N];

for (unsigned i = 0; i < N; i++)
    numbers[i] = rand();
```

В стандарте C++20 введена библиотека математических констант `numbers`:  
<https://en.cppreference.com/w/cpp/numeric/constants>.

```
#include <numbers>
using std::numbers::pi;

float pizza_area = pi * 35*35;
```

Вместо создания собственных констант стоит использовать аналогичные константы из математической библиотеки.

См. также [6] о константах.

todo: constexpr

### 1.2.3 Перечисления (enum)

Плохая практика – вводить не именованные условные обозначения. На-пример

```
/// выводит текст на экран.
/// color = 0, 1, 2 -- обознач. цвет текста  красный, зелёный, синий
void print_text( string text, int color){
    // ...
}
```

Такие обозначения не несут прямого смысла, приходится запоминать или постоянно обращаться к документации. Это лишняя возможность допустить ошибку. Один из вариантов решения – ввод констант

```
const short COLOR_RED = 0;
const short COLOR_GREEN = 1;
const short COLOR_BLUE = 2;
```

Но в языке C++<sup>2</sup> есть специальный тип данных, который помогает вводить наборы однотипных обозначений – *перечисления* (enum).

---

<sup>2</sup>как и во многих других языках программирования

```
// объявление типа данных перечисление
enum Color {
    // и его значения для обозначения цветов:
    Red, Green, Blue
};
```

Значения типа enum неявно конвертируются в целочисленный тип, например при выводе в консоль. Можно говорить, что перечисление – это набор целочисленных констант. Первое в объявлении значение равно 0, второе 1 и так далее.

Пример объявления переменных

```
// эти способы инициализации переменных равнозначны
Color c1 = Red;
Color c2 = Color::Red;
```

Перечислимый тип может использовать в операторе выбора

```
// пример использования
switch(r){
    case red : std::cout << "red\n"; break;
    case green: std::cout << "green\n"; break;
    case blue : std::cout << "blue\n"; break;
}
```

Приведём улучшенный вариант функции из примера выше

```
/// выводит текст на экран. color -- цвет текста
void print_text( string text, Color color){
    // ...
}
```

См. также табличные методы [6].

todo: enum class, or enum struct

Документация: [en.cppreference.com/w/cpp/language/enum](http://en.cppreference.com/w/cpp/language/enum)

## 1.2.4 auto, typedef

Указание **auto** вместо типа заставляет компилятор самостоятельно под- определение типа  
 ставить тип ориентируясь на задаваемое значение. статическое, то есть до  
 запуска программы!

```
auto x = 20;           // int
auto y = 3.14159;      // float
```

```
auto z = "gues type";    // char*
auto a; // ошибка! Не задано значение!
```

Использовать auto может быть для повышения читаемости кода, когда имя типа длинное и не принципиально для понимания кода. Однако, в учебных целях, стоит всегда оставлять комментарии о выводимых типах данных.

todo: auto и typedef внутри функций

## 1.2.5 Преобразования типов

Преобразования строкового типа

```
string s = std::to_string(123);
float f1 = stof("123.5");
```

см также stoi, stod и т.п.

[en.cppreference.com/w/cpp/string/basic\\_string/to\\_string](http://en.cppreference.com/w/cpp/string/basic_string/to_string)

Для преобразования вещественных чисел в строку см. класс для работы со строковыми потоками stringstream в разделе 3.2 string stream и функцию format, добавленную в стандарте C++20.

**Типобезопасность**(type safety) –

**C-style cast** – ...

...

**static\_cast** безопасно (с проверкой соответствия типа) преобразовывает выражение одного типа в другой:

...

Пример:

```
static_cast < new_type > ( expression )
```

**Неявное преобразование типов** todo:

```
short a = 42;
int b = a;
```

документация по функции format  
[en.cppreference.com/w/cpp/uti](http://en.cppreference.com/w/cpp/utility/string/basic_string_format)  
Для преобразования чисел в строку модно использовать функцию sprintf, перешедшую в C++ из C:  
[cplusplus.com/reference/cstdio](http://cplusplus.com/reference/cstdio)

на момент декабря 2022 года библиотека format поддерживается G++13 и MSVC 19

см. также раздел 5.4 Динамический полиморфизм и оператор dynamic\_cast, который выполняет проверку возможности преобразования типов на этапе выполнения программы

```
int c = 12354;
short d = c;
```

Переполнение типа: todo

```
char a = 256;    // a = 0
char b = 42, c = 230;

short b + c;      // ok
todo: max_int etc
```

см. также преобразование объектов: ...

### 1.2.6 Приоритет операторов

*todo: ...*

### 1.2.7 Ввод и вывод на экран

#### Вывод

Переменные и функции для ввода и вывода объявлены в заголовочном файле `iostream`.

```
#include <iostream>
```

```
using namespace std;
```

```
cout << "Hello, World!";
```

```
// endl - вывод символа конца строки и очистка буфера вывода
cout << "Hello, World!" << endl;
```

```
// Вывод переменной
```

```
float x;
cout << x << endl;
```

```
// Вывод данных нужно подписывать
```

```
cout << "x = " << x << endl;
```

`cout` – объект предназначенный для вывода на стандартный вывод. Этот объект содержит оператор `<<` для вывода данных в консоль. Левый операнд этого оператора – объект `cout`, правый операнд – выводимые данные.

## Форматирование

```
#include <iomanip>

cout << 3000000.14159265 << " "; // вывод: 3e+06;

// 12 позиций на всё число; человекочитаемый формат (без e); два знака после запятой
cout << setw(12) << fixed << setprecision(2);

cout << 3000000.14159265 << " "; // вывод: 3000000.14; (2 пробела в начале)
```

## Форматирование с преобразованием в строку (начиная со стандарта C++20)

```
# include <string>
// преобразование числа в строку с помощью функции форматирования строки
// {:.3f} - формат вывода вещественного (f) числа с 3 знаками после запятой
string s = format("{:.3f}", 3000000.14159265);
```

Подробнее о форматировании в документации:

[https://en.cppreference.com/w/cpp/utility/format/formatterStandard\\_format\\_specification](https://en.cppreference.com/w/cpp/utility/format/formatterStandard_format_specification)

## Форматирование больших чисел с добавлением разделителя для разрядов

```
cout << 123456789 << endl;
cout.imbue(std::locale(""));
cout << 123456789 << endl;
```

imbue включит вывода в соответствии с переданными настройками локализации для текущего потока (в примере это cout). std::locale("") – создание объекта, представляющего настройки локализации ОС.

объект – экземпляр составного типа данных – класса

Вывод программы:

```
123456789
123 456 789
```

Разряды могут быть разделены не только пробелами, а например запятыми, в зависимости от настроек ОС.

Получить название текущих настроек локализации, для потока вывода cout:

```
cout << cout.getloc().name();
```

Пример вывода:

```
LC_CTYPE=en_US.UTF-8;LC_NUMERIC=ru_RU.UTF-8;LC_TIME=ru_RU.UTF-8;  
LC_COLLATE=en_US.UTF-8;LC_MONETARY=ru_RU.UTF-8;LC_MESSAGES=en_US.UTF-8;  
LC_PAPER=ru_RU.UTF-8;LC_NAME=ru_RU.UTF-8;LC_ADDRESS=ru_RU.UTF-8;  
LC_TELEPHONE=ru_RU.UTF-8;LC_MEASUREMENT=ru_RU.UTF-8;LC_IDENTIFICATION=ru_RU.UTF-8
```

## Ввод

`cin` – объект предназначенный для чтения данных с клавиатуры, объявлен  
в `iostream`

`<<` – оператор чтения данных с клавиатуры.

Левый операнд – объект `cin`;

Правый операнд – переменная.

```
float x;  
cin >> x;
```

Посимвольный ввод

```
char c = '\0';  
while (cin.get(c)) {  
    // ...  
}
```

<https://en.cppreference.com/w/cpp/utility/format/format>

<https://www.cplusplus.com/reference/iostream/setprecision/>

Про обработку ошибок при вводе значений идет речь в параграфе 2.1

Пример обработки исключений при консольном вводе.

## 1.2.8 Операторы

**Оператор** –

**Операнд** –

## 1.2.9 Оператор присваивания

Оператор присваивания в C++ возвращает присвоенное значение. Напри-  
мер

```
int a, b;  
a = b = 7;
```

```
// для наглядности, можно представить выражение так
// a = ( b = 7 );
// a = ( 7 ); - после выполнения самого правого оператора присваивания.
```

Сначала выполнится оператор присваивания (`b = 7`), так как он стоит в правой части выражения `a = ....`

Эта особенность оператора присваивания может приводить к ошибкам в операторах, где вместо оператора сравнения `==` случайно записан оператор `=`. См. пример с условным оператором ниже.

### 1.2.10 Логические операции

`&&` – И

`||` – ИЛИ

`!` – Не

### Битовые операции

операторы!битовые `&` – И

`|` – ИЛИ

`~` – НЕ

`^` – XOR

С помощью битовых операций можно получить значения отдельных байт в целом числе

```
int nint;
char byte1, byte2, byte3, byte4;

byte1 = nint & 0x000000ff
byte2 = (nint & 0x0000ff00) >> 8
byte3 = (nint & 0x00ff0000) >> 16
byte4 = (nint & 0xff000000) >> 24
```

### 1.2.11 Тернарный условный оператор

**Тернарный оператор** – оператор с тремя операндами. В C++ такой оператор приспособлен для сокращённой форму записи условного оператора.

`a ? b : c`

Если условие *a* верно, то выполняется *b*, если нет, то *c*.

Пример нахождения максимального числа из *x* и *y*:

```
float a, b, max;  
// ...  
max = (x>y) ? x : y
```

todo: пример без присваивания

Документация: [en.cppreference.com/w/cpp/language/operator\\_other](http://en.cppreference.com/w/cpp/language/operator_other)

## 1.2.12 Управляющие операторы

### Условный оператор

*todo*

В этом условном операторе вывод на экран никогда не случится. В условии вместо оператора сравнения `==` записан оператор присваивания.

```
int x = 0;  
if ( x = 0 ){  
    cout << "x is zero";  
}
```

После присваивания значения выражение `x = 0` преобразуется в `0`, что далее будет приведено к логическому типу, в данном случае к значению `false`. Таким образом записанное условие всегда ложно.

Чтобы избежать подобных ошибок можно записывать литерал (значение) слева от оператора `==`:

```
int x = 0;  
if ( 0 == x ){  
    cout << "x is zero";  
}
```

Смысл логического выражения не поменялся, но если пропустить один символ `=` то ошибка сразу станет заметна ещё на этапе компиляции.



Оператор выбора

Оператор цикла while

Оператор цикла do while

Оператор цикла for

Цикл по коллекции

```
for (type &x: array) {  
    ...  
}
```

Пример.

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

```
for(int x : my_array)  
    cout << x << " ";
```

*// в x записывается ссылка на элемент, можно изменять массив*

```
for(int &x : my_array)  
    x = x*x;
```

```
int *d_array = new int[5];
```

*// ошибка! число элементов массива не известно*

```
for(int x : d_array)  
    cout << x << " ";
```

см. также функцию `for_each` из стандартной библиотеки алгоритмов.

## 1.3 Динамическая память, указатели и ссылки

### 1.3.1 Указатели

**Указатель** (pointer) – переменная, диапазон значений которой состоит из адресов ячеек памяти или специального значения – нулевого адреса (**nullptr**).

Другими словами, указатель – тип данных, способный хранить адрес памяти.

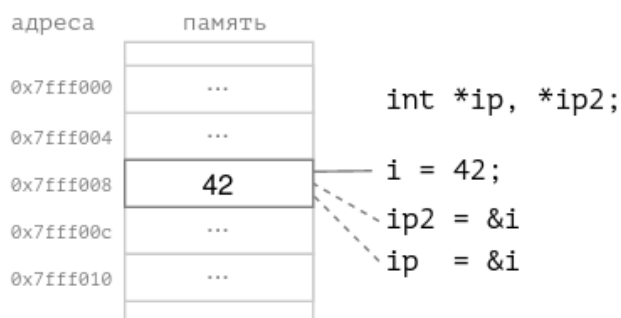


Рис. 1.4. Условное представление памяти: переменная *i*, указатели *ip* и *ip2*, хранящие адрес этой переменной

При объявлении указателя после типа данных, на который он должен указывать, ставится \*

```
// объявление указателя на тип int
int * ip;
// объявление указателя на тип float, инициализация пустым адресом
float *fp = nullptr;
```

до C++11 вместо **nullptr** использовался идентификатор **NULL**, который был определён директивой препроцессора:  
**#define NULL 0**

обращение к не инициализированному указателю приводит к неопределённому поведению

Основные операции для работы с указателями:

- взятие адреса (оператор &), используется при записи адреса переменной в указатель;
- разыменованье (оператор \*) – обращение к значению, адрес которого записан в указателе.

Инициализация указателя значениями (литералами)

```
// объявления и инициализация указателя
int * ip0 = 0;

// вместо 0 рекомендуется использовать nullptr
// объявление указателя на тип int, инициализация нулевым указателем
int * ip = nullptr;
```

```
// Ошибка: любые, отличные от 0, числовые значения для указателя недопустимы
ip = 1235489;
```

## Указатели и адреса переменных

```
int i = 42;

// в указатель можно записать адрес переменной
// для этого используется оператор взятия адреса &
ip = &i;

// теперь можно обращаться к переменной i через указатель.
// используем разыменование (оператор *) чтобы обратиться не к адресу,
// который записан в указателе, а к значению, на которое он указывает
*ip = 8; // переменная i теперь содержит 8

int *ip2;

// можно записывать в один указатель другой
// если типы данных, на которые они ссылаются, совпадают
ip2 = ip;
// *ip2 = 8
// *ip = 8
// i = 8

*ip2 = 1950;
// *ip = 1950
// i = 1950
```

В итоге имеем одну переменную `i` типа `int`, на которую в конце концов указывает два указателя `ip` и `ip2`.

## Указатель на пустой тип (`void *`)

В C широко используется указатель на пустой тип (`void *`) для передачи разнородных данных в функции.

### Пример

```
int x = 0x0a0b0c0d;           // int занимает 4 байта

// небезопасное преобразование типа:
void * vp = (void*) &x;
// теряется инф-я о размере области памяти, на которую указывает vp

// небезопасное преобразование типа:
char *bytes = (char*)vp;

// вывод
```

здесь используется преобразование типов C-style cast; для

преобразования типов одинакового размера (в том числе в массивы) см. union

```
cout << (int)bytes[0] << " " << (int)bytes[1] << " "
      << (int)bytes[2] << " " << (int)bytes[3];
```

Результат: 13 12 11 10

В C++ рекомендуется избегать использование указателей на пустой тип, если есть альтернатива.

### 1.3.2 new и delete

todo:

malloc vs new

висячие указатели

утечка памяти

### 1.3.3 Ссылки

**Ссылки** (reference) похожи на указатели, только с разницей

- Ссылка не может менять своё значение
- Следовательно при объявлении ссылки она обязательно инициализируется
- При обращении к значению по ссылке оператор \* не требуется
- Для взятия адреса другой переменной оператор & не требуется

Про ссылку можно думать как про другое имя для объекта

```
int i = 42;
// при _объявлении_ ссылки используется &
// здесь не стоит путать с оператором & для взятия адреса,

int &il = i;

// оператор разыменования не требуется
int n = il;
il = 100;    // обращение к ссылке как к "нормальной" переменной
// i = 100; n = 42

// ссылка не может указывать на литерал
int &il2 = 100;    // ошибка!
```

См. также параграф 3.3 «Умные указатели» о типах данных, автоматически очищающих выделенную память.

### 1.3.4 Спецификатор const и указатели

Указатель на константу:

```
const int N = 512, M = 1024;  
int A = 42;
```

нет разницы, где  
указывать модификатор  
const до имени типа или  
после

```
    int *p1 = &N; // Ошибка: можно хранить адрес только переменной int!  
const int *p2 = &N; // указатель на константу  
int const *p3 = &N; // аналогичный указатель на константу  
p2 = &M;           // можно менять указатель  
*p2 = 256;         // Ошибка: нельзя менять константу!
```

```
p1 = &A;  
*p1 = 43;
```

Если модификатор **const** стоит справа от указателя, то он относится к указателю.

```
int *const p4 = &N;
```

Подобные объявления нужно читать справа налево: константный (const) указатель (\*) на константу (int const).

Следующее объявление можно понимать как константный указатель **\*const** на указатель на целое (**int \***).

```
int * *const p5 = ...;
```

Это может быть и массивом неизменных указателей (\*const), каждый элемент которого может указывать (\*) на одно или несколько значений целого типа (int).

см. раздел про динамические массивы

todo: константы + ссылки

### 1.3.5 Проблемы динамической памяти

перезаписывание указателя

потеря указателя

Утечка памяти (memory leak)

неосвобождённая память

см. также раздел про умные указатели

## 1.4 Функции

Короткий общий вид объявления (declaration) функции:

```
возвр.тип func_name( тип параметр, ... ); // заголовок функции
```

Короткий общий вид определения (definition) функции:

```
возвр.тип func_name( тип параметр, ... ) // заголовок функции
{
    // тело функции
    return выражение-с-типом; // не обязательно*
}
```

В некоторых компиляторах возврат значения обязателен, если тип возвращаемого значения не пустой (void). Если возвращаемый тип функции **void**, то после **return** не должно быть никакого выражения.

Более полный вид определения функции см. в документации:  
[en.cppreference.com/w/cpp/language/function](http://en.cppreference.com/w/cpp/language/function) *Function definition*.

Функцию можно объявить заранее, это сделает возможным её использование, а определить позже. Часто функции объявляют в заголовочных (.h) файлах, а определяют в cpp файлах.

**Формальные параметры** – параметры, приведённые в заголовке функции.

**Фактические параметры** – параметры переданные при вызове функции, подставляются на место формальных параметров.

Формальными параметры представлены переменными. Фактические параметры могут быть переменными, литералами (значениями) или выражениями

```
// x,s -- формальные параметры
float foo( int x, string s ) {
    cout << s << x; }

int main(){
    // вызов функции
    foo( 1, "qwerty");
    // 1, "qwerty" -- фактические параметры

    foo( 2+2, "qwerty");
    foo( rand(), "qwerty");
}
```

```

// ошибка: тип первого параметра не может быть преобразован в int
foo( "abc", "qwerty");

int y;
string str;
foo( y, str);          // y, str -- фактические параметры
}

```

Типы формальных параметров и соответствующих им фактических должны либо совпадать, либо типы фактических параметров должны приводиться к типу соответствующих формальных.

```

foo( 1/7, "qwerty");      // 1/7 -> 0 (int)

foo( 8.73121, "qwerty");  // 8.73121 -> 8 (int)
}

```

## Возврат значения из функции

```

float foo( int x ) {
    return rand() % x; }

// Функция не возвращающая ничего
void bar( int x) {
    cout << rand() % x << endl; }

int main(){
    // вызов функции возвращающей значение
    int a = foo(10);
    cout << foo(10);
    int b = 5 + foo(10);
    foo(10);          // корректно, но возвращаемое значение потеряно

    bar();
}

```

Если функция не должна возвращать значений (возвращаемый тип void), то оператор return просто завершает выполнение функции.

```

void foo() {
    cout << "1";
    return;
    // функция никогда не выполнит операцию:
    cout << "2";
}

```

См. примеры функций с аргументами массивами в разделе 1.6.4 Динамические массивы.

todo: Принцип единственной ответственности

## Параметры-ссылки, параметры-значения и параметры-константы

Для фактического параметра переданного *по значению* внутри функции создаётся локальная копия. Изменение этой копии (формального параметра) не влияет на фактический параметр.

```
int a = 42;

// x - формальный параметр-переменная
void foo ( int x ) { x = 123; }

foo( a ); // a - фактический параметр
cout << a; // 42
// переменная a не изменилась
```

Для фактического параметра переданного в функцию *"по ссылке"* на самом деле передаётся его *адрес*. Значит изменения формального параметра внутри функции означают изменения фактического параметра.

```
// x - формальный параметр-ссылка
void foo ( int &x ) { x = 123; }

int a = 42;
foo( a ); // a - фактический параметр
cout << a; // 123
// переменная a изменилась
```

Для изменения фактического параметра внутри функции можно сделать формальный параметр не ссылкой, а указателем. Однако это менее удобно.

```
// x - формальный параметр-указатель
void foo ( int *x ) {
    // требуется разыменование
    *x = 123;
}

int a = 42;
foo( &a ); // a - фактический параметр; требует операция обращения к адресу
cout << a; // 123
// переменная a изменилась
```

Но такой способ передачи данных в функцию хорошо подходит для массивов (см. раздел: 1.6.4 Динамические массивы)

Переменные, которые занимают достаточно много памяти (классы, структуры, объединения) стоит передавать по ссылке, чтобы избежать создания их копии при вызове функции.



**Параметры-константы.** Если такая переменная не должна менять значение внутри ссылки, то используйте модификатор **const**:

```
struct Coordinate{
    double latitude;
    double longitude;
};

void print_coordinate( const Coordinate& c){
    c.latitude = 51;    // ошибка!
    cout << c.latitude << ", " << c.longitude;
}

int main(){
    Coordinate c1{-19.949156, -69.633842};
    print_coordinate(c1);
}
```

## Значения параметров по умолчанию

Когда параметр необходим, но функция часто вызывается с определённым его значением, то можно задать для него значение по умолчанию.

```
void foo( int y = 1950 ) {cout << x;}

foo( 123 ); // 123
foo()      // 1950
```

Формальные параметры со значением по умолчанию должны быть последними.

- Используйте для аргументов, значения которых часто принимают одно и то же значение
- Приводите эти аргументы в последнюю очередь
- Не используйте неожиданных значений по умолчанию

## Перегрузка функций (function overloading)

Функциям выполняющие одинаковую работу с разными по типу наборами данных можно давать одинаковые имена. Компилятор определит по набору фактических параметров, какая функция должна быть вызвана.

```
void foo(int x){ cout << "Перегрузка";}
```

```

void foo(float x){ cout << "Overloading";}

void foo(int x, int y){ cout << "Überanstrengung!!!";}

foo(20);      // Перегрузка
foo(20.0);    // Overloading
foo(1, 2);    // Überanstrengung!!!
foo(1, 2.0)   // Überanstrengung!!!

```

- Функциям выполняющим одинаковую работу с разными данными можно давать одинаковые имена
- Перегруженные функции должны отличаться по типу и количеству параметров
- Перегруженные функции не отличаются по типу возвращаемого значения
- При компиляции перегруженным функциям даются разные имена.
- Какая из перегруженных функций будет вызвана также определяется на этапе компиляции

#### 1.4.1 Перегрузка функций

Функциям выполняющие одинаковую работу с разными по типу наборами данных можно давать одинаковые имена. Компилятор определит по набору фактических параметров (но не по типу возвращаемого значения), какая функция должна быть вызвана.

```

void foo(int x){ cout << "Перегрузка";}
void foo(float x){ cout << "Overloading";}
void foo(int x, int y){ cout << "Überanstrengung!!!";}

foo(20);
// Перегрузка
foo(20.0); // Overloading
foo(1, 2); // Überanstrengung!!!
foo(1, 2.0) // Überanstrengung!!!

```

Функциям выполняющим одинаковую работу с разными данными можно давать одинаковые имена.

Перегруженные функции должны отличаться по типу и количеству параметров.

Перегруженные функции не отличаются по типу возвращаемого значения.

При компиляции перегруженным функциям даются разные имена. Решение о том, какой вариант функции должен быть вызван

Какая из перегруженных функций будет вызвана также определяется на этапе компиляции.

todo: Алгоритм поиска реализации перегруженной функции.

#### 1.4.2 Значения параметров по умолчанию

#### 1.4.3 inline-функции

...

#### 1.4.4 static функции и локальные переменные

static функции доступны только в своей единице трансляции (cpp файле, в котором приведены или в который включились директивой include). Если одна и та же статическая функция определена в разных cpp файлах, то при компиляции не возникнет ошибка множественного определения (multiple definition).

Статическая локальная переменная хранится как глобальная переменная, инициализируется при первом вызове своей функции, сохраняет свою локальную область видимости.

#### 1.4.5 Спецификатор constexpr

**constexpr** –

#### 1.4.6 Выводы и рекомендации

- Функции делают возможным алгоритмическую декомпозицию
- Функции делают возможным повторное использование кода
- Для того чтобы пользоваться функцией не нужно обладать минимальными знаниями о её внутреннем устройстве
- Легче повторно использовать функцию служащую одной цели
- Следует стремиться к чистоте функций
- Стоит избегать использования глобальных переменных в функциях
- Параметры, которые дорого копировать следует передавать по ссылке

- Параметры, переданные по ссылке, но не изменяющиеся в теле функции нужно делать константными.

## Документирующие комментарии

```
// плохо:  
// функция вычислений; возвращает float  
float bmi(float m, float h);  
  
// лучше:  
// вычисляет индекс массы тела  
float bmi(float m, float h);  
  
// хорошо:  
// вычисляет индекс массы тела по массе (m) в кг. и росту (h) в метрах  
// бросает исключение invalid_argument если h==0  
float bmi(float m, float h);  
  
// отлично (машинно-читаемый комментарий для  
// системы документирования Doxygen):  
/// вычисляет индекс массы тела;  
/// бросает исключение invalid_argument если h==0  
/// \param m масса тела в кг.  
/// \param h рост в метрах  
/// \return индекс массы тела  
float bmi(float m, float h);
```

todo: Doxygen – система документирования для языков C++, Си, Python, Java, C#, PHP и др.

См. также параграф Самодокументируемый код в [6].

См. также параграф 2.2 Функции.

## 1.5 Пространства имён (namespaces)

Объявление пространства имён:

```
namespace имя {  
...  
}
```

todo: Безымянные пространства имён ::

Оператор **using** может использоваться для включения указанного пространства имён в текущее пространство имён.

Синтаксис использования:

```
using имя_пи::member_name;  
// теперь можно обращаться к только member_name непосредственно,  
// без префикса имя_пи::  
  
using namespace имя_пи;  
// теперь можно обращаться ко всем именам, описанным в имя_пи, непосредственно,  
// без префикса имя_пи::
```

Одно и то же пространство имён можно дополнять сколько угодно раз. Как в рамках одного файла исходных кодов, так и нескольких. Пример такого кусочного объявления — пространство имён `std`. Оно описано в разных файлах, например `vector` и `string`.

Обычно одно и то же пространство имён описывается в логически соответствующих друг другу заголовочном и `cpp` файле.

`geometry.h`

```
namespace geometry{  
    /// вычисляет площадь треугольника по сторонам  
    float triangle_square(float a, float b, float c);  
  
    // ...  
}
```

`geometry.cpp`

```
namespace geometry{  
    /// вычисляет площадь треугольника по сторонам  
    float triangle_square(float a, float b, float c){  
        // определение функции  
    }  
}
```

Аналогично определить члены пространства имён можно и указывая перед имя самого пространства имён: `geometry.cpp`

```
/// вычисляет площадь треугольника по сторонам  
float geometry::triangle_square(float a, float b, float c){  
    // определение функции  
}
```

такой способ используется при определении методов класса, где вместо имени простого пространства имён используется имя класса

Одинаковые переменные, объявленные в разных пространствах имён (даже в одном файле) не создают конфликта имён.

Пространство имён помогают логически объединить схожие типы, функции константы и переменные. В одной файле может быть описано сколько

угодно пространств имён, в том числе сложенных в друг друга. Это поможет отделить разные по смыслу участки кода.

## 1.6 Массивы

### 1.6.1 Статические массивы

Объявление одномерного массива:

```
тип_элемента_массива идентификатор [ размер ];  
тип_элемента_массива идентификатор [ размер ] = {значения эл-в массива через запятую};
```

Объявление многомерных массивов:

```
тип_элемента_массива идентификатор [ размер1 ][ размер2 ];  
тип_элемента_массива идентификатор [ размер1 ][ размер2 ] =  
    { {...}, {...}, {...}, ... };  
  
тип_элемента_массива идентификатор [ размер1 ] ... [ размерN ];  
тип_элемента_массива идентификатор [ размер1 ] ... [ размерN ] =  
    { { ... {...} ... }, ... };
```

При инициализации многомерного массива вложенность скобок со значениями должна совпадать с размерностью массива. Количество пар фигурных скобок со значениями – количеству элементов по каждой размерности (см. пример инициализации массива `matrix` ниже). Тип элемента массива может быть любым (просто тип, указатель, структура, класс ...) кроме `void` и ссылочных типов.

Пример объявления и использования массива:

```
int a[128];  
int b[256];  
  
// обращение к элементу по его индексу  
a[0] = 42; // нумерация с нуля
```

в разделе 4.3.2 приведены примеры работы с массивами из объектов – экземпляров составного типа – классов

Тип таких массивов описывается как `Типе[N]`, т.е. содержит количество элементов. Два статических массива с одинаковым типом элементов но разным их количеством (а и b в примере) имеют разный тип.

```
cout << sizeof(a) << "\n";           // 512  
cout << sizeof(*a) << "\n";          // 4    (размер int)  
cout << sizeof(a)/sizeof(*a) << "\n"; // 128
```

Массивы, как и переменные остальных типов в C++, автоматически не инициализируются. Но можно вручную задать значения всех элементов:

```

int a[128] = {0};           // инициализация всего массива нулями
int b[5] = {1,2,3};        // результат инициализации: 1, 2, 3, 0, 0

int matrix[2][3] = { {1,2,3},
                      {4,5,6} };

int days[12] = {31, 27, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

// если задаётся список значений, то их количество можно не указывать
int days1[] = {31, 27, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

```

Обращение к несуществующим индексам массива (например `a[128]`) в стандарте регламентируется как *неопределённое поведение* (undefined behavior, UB). Программа может аварийно завершиться или продолжить выполнение с непредсказуемыми последствиями. Например, если в оперативной памяти сразу после массива `a` будет располагаться массив `b`, то его первый элемент изменится:

```

int a[8] = {0};
int b[8] = {0};
// выстреливаем себе в ногу:
a[8] = 111;
b[-1] = 222;
cout << a[7] << "; " << b[0];           // 222; 111

```

Хорошей практикой считается задавать размер массива с помощью константы, а не через *магическое число* (magical number). Это упрощает модификацию и понимание кода.

```

// храните размер статического массива в константе
unsigned const N = 128;
float t[N];
t[N-1] = 36.6; // последний элемент массива

for (int i = 0; i < N; i++)
    cout << t[i];

```

Переменные, используемые для работы со статическими массивами, фактически являются указателями. Поэтому прямое обращение к ним выдаст адрес, где находится первый элемент массива

```

cout << "days_addr = " << days_c << "\n"; // 0x7ffc364faf90
// смещение на один элемент (int, 4 байта) относительно адреса начала массива
cout << "days_addr+1 = " << days_c+1 << "\n"; // 0x7ffc364faf94

```

Доступна и операция разыменования



```
cout << *days_c;           // 31

cout << *(days_c+1);       // 27
// аналогично:
cout << days_c[1];          // 27
```

### 1.6.2 Массив из символов

В C++ строки представляются массивом из символов. Но для удобства записи этот массив можно инициализировать строковым литералом (значением), не перечисляя отдельные символы через запятую:

```
char str1 [] = "Hello";
```

В символьных массивах принято обозначать конец строки символом с кодом 0 (обозначается '\0'). При инициализации строкой этот служебный символ добавляется в строку автоматически. Если в конце не поставлен ноль, то поведение программы при обращении к такому массиву не определено (undefined behavior).

Массив из примера выше можно инициализировать посимвольно:

```
char str2 [] = {'H', 'e', 'l', 'l', 'o', 0};

cout << sizeof(str1);           // -> 6 (5 символов строки и нулевой символ)
cout << sizeof(str2);           // -> 6 (5 символов строки и нулевой символ)
```

см. также класс string.

### 1.6.3 Функции для работы с массивами

Функции копирования (strcpy) и др. функции преобразования строковых массивов: [en.cppreference.com/w/cpp/header/cstring](http://en.cppreference.com/w/cpp/header/cstring).

Функции копирования и перемещения участков памяти, объявленные в файле cstring:

```
void* memcpy( void* dest, const void* src, std::size_t count );

void* memmove( void* dest, const void* src, std::size_t count );
```

dest – приёмник, src – источник, count – количество байт для копирования или перемещений. Эти функции обеспечивают наилучшую производительность для своих задач.

Пример копирования данных из массива arr1 в arr2:

```
float arr1[] = {1,2,3,4,5,6,7,8,9,0};
float arr2[10] {99};

memcpy(arr2, arr1, 10 * sizeof(float));

for (unsigned i = 0; i < 10; i++)
    cout << arr2[i] << " ";
```

Результат работы программы:

1 2 3 4 5 6 7 8 9 0

### 1.6.4 Динамические массивы

Общий синтаксис объявления динамического массива с выделением памяти:

```
тип_элемента_массива * идентификатор = new тип_элемента_массива [ размер ];

тип_элемента_массива * идентификатор =
    new тип_элемента_массива [ размер ] {начальные значения через запятую};
```

Адрес начала динамического массива в памяти хранится указателе (как и для одиночного значения), а память под них выделяется оператором **new** в куче во время выполнения программы (динамически).

**Пример.**

```
unsigned n = 128;

// выделение памяти под n значений типа int (массив)
int *a = new int[n];
int *b = new int[n] {0};    // инициализация массива нулями
int *c = new int[n] {17, 29}; // инициализация массива: 17, 20, 0, ..., 0

// не массив, указатель на одно значение типа int
int *z = new int;

// обращение к элементам такое же как и для статического массива
a[0] = 42;
int x = a[2];
// аналогично
x = *(a+2);

// после окончания работы с массивом обязательно освобождаем его память
delete[] a;
delete[] b;
delete[] c;
```

При том, что в C++ нельзя через указатель узнать количество памяти занимаемой массивом, операция `delete[]` а освободит ровно такое количество памяти, какое занимает весь массив. Это количество изначально сохраняется при вызове оператора `new` и хранится в памяти прямо перед данными.

В отличие от одиночных значений, хранящихся в куче, освобождать память занимаемую массивом нужно оператором `delete[]` а;. Вызов оператора `delete` для массива не считается синтаксической ошибкой, но освобождает только память занимаемую указателем.

В статическом массиве размер был частью типа, поэтому было возможно с помощью оператора `sizeof` вычислить размер массива. Так как переменная динамического массива не отличима от указателя, для динамических массивов аналогичное вычисление размера невозможно:

```
int *a = new int[ rand() ];           // массив случайного размера

cout << sizeof(a) << "\n";           // 8    (размер указателя)
cout << sizeof(*a) << "\n";          // 4    (размер int)

// эта операция не имеет смысла:
cout << sizeof(a)/sizeof(*a) << "\n"; // 2
```

Поэтому для каждого динамического массива программист должен сохранять размер в отдельной переменной.

**Указатель vs динамический массив vs статический массив из указателей:**

```
int a;

// указатель
int *y;
y = &a

// статический массив из 128 указателей:
int * x[128];
x[0] = &a;
x[1] = new int; // выделение памяти под одно значение типа int

// динамический массив
int *z = new int[128]
```

**Передача массивов в функции.** Статические массивы в функции передавать проблематично, из-за того что их тип содержит информацию о количестве элементов. А значит функция будет способна принимать массив только одного фиксированного размера.

Динамические массивы в функции передаются как указатели. При этом нужно передавать размер через отдельный параметр.

```
void array_rnd_fill(int* arr, unsigned n){
    for (unsigned i = 0; i<n; i++)
        arr[i] = 1;
}

int sum_array(const int * arr, unsigned n){
    // const int * -- массив из констант, запрещает изменение элементов массива
    int s = 0;
    for (unsigned i = 0; i<n; i++)
        s += arr[i];
    return s;
}

int main(){
    unsigned n = 20;
    int *a = new int[ n ];
    array_rnd_fill(a, n);
    cout << sum_array(a, n);
}
```

Хорошая практика: передавать в функцию массив, где он не должен изменяться, через константный формальный параметр. Он запретит непреднамеренное изменение элементов массива внутри функции.

При передаче массива в функцию `array_rnd_fill`, формальный параметр `arr` будет содержать *копию* адреса, где расположен массив.

```
void foo(int* arr, unsigned n){
    // изменение элементов массива -- это изменение фактического параметра
    arr[0] = 10;
    *(arr+1) = 20;    // изменение второго элемента массива
    arr = nullptr;    // изменение формального параметра массива (адреса)
}

int main(){
    unsigned n = 3;
    int *a = new int[n] {0};
    foo(a, n);
    a == nullptr;    // false
    // a = {10,20,0}
}
```

## Возврат массивов из функций

```
// функция выстреливает в ногу
int* bar(){
    int arr[128];
```

```

    // логическая ошибка: возврат указателя на локальную переменную
    // после завершения функции память, на которую указывает arr освободится
    return arr;
}

// возвращает массив случайного размера n
// не выстреливает в ногу
int* foo(unsigned &n){
    n = rand()+1;
    int* arr = new int[n];
    return arr;
}

int main(){
    unsigned n;
    int *a = foo(n);
    int *b = bar();
    // b ссылается на область памяти, которая уже освобождена
    b[0] = 42;          // Undefined behavior!
}

```

## Двумерные динамические массивы

```

#include <iostream>
#include <iomanip>          // для настроек ввода и вывода

using namespace std;

/// выводит двумерный массив (матрицу) arr размерности rows x cols на экран
void print_matr(int** arr, unsigned rows, unsigned cols){
    // в функцию передаётся указатель на массив из массивов
    // поэтому его элементы можно изменить при желании
    for (int i = 0; i < rows; ++i){
        for (int j = 0; j < cols; ++j)
            // вывод числа в поле шириной 11 символов
            cout << setw(11) << arr[i][j] << " ";
        cout << "\n";
    }
}

int main(){
    const unsigned N = 3;          // число строк
    const unsigned M = 4;          // число столбцов

    // выделение памяти под двумерный массив:
    int * *matr = new int*[N];    // память под массив указателей (массивов)
    // выделение памяти под двумерные массивы (строки матрицы)
    for (int i = 0; i < N; ++i)
        matr[i] = new int[M];    // память под отдельные массивы
    // matr[i] можно воспринимать как строки матрицы

    print_matr(matr, N, M);
}

```

```

// освобождение памяти
for (int i = 0; i < N; ++i)
    delete[] matr[i];
delete[] matr;

return 0;
}

```

## 1.7 Устройство памяти программы

- Статическая память (data на рис.). Хранит глобальные переменные.
- Динамическая память (heap, куча). Память, которая выделена оператором **new**.
- Автоматическая паять (stack, стек). Хранит адреса возврата функций, локальные переменные, параметры функций.
- Сегмент кода (text на рис.)

не стоит путать область памяти *стек*, с одноимённым типом данных и стеком процессора

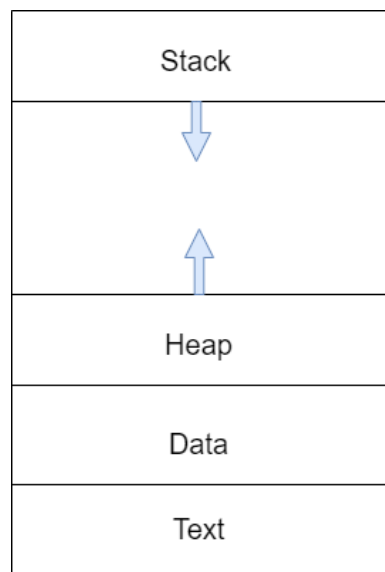


Рис. 1.5. Устройство памяти программы

Использование оператора **new** внутри функции приводит к выделению памяти в куче. Но стоит дополнительно позаботиться, чтобы не потерять указатель на эту память, который может быть локальной переменной.

```

void foo(){
    int * arr = new int [1024];
    // arr - указатель, локальная переменная находится в стеке.
    // 1024 элемента массива располагаются в куче
}

```

```

    // doing stuff ....
}

```

После завершения функции `foo` исчезнет переменная `arr`. Уже будет нельзя ни освободить эту память, ни обратиться к ней то есть произойдёт утечка памяти. Можно решить проблему вернув адрес указателя:

```

int* foo(){
    int * arr = new int [1024];

    // doing stuff ...

    return arr;
}

```

```

int * array = foo();

// ...

delete[] array;

```

**access violation, segmentation fault** –

**Стек вызовов** (call stack) – ...

## 1.8 Заголовочные файлы и `.cpp` файлы

До стандарта C++20 в языке не существовало полноценных модулей. Модулем обычно называлась пара файлов: заголовочный (`.h`) и `.cpp` файл. Для удобства таким файлам дают одинаковые (с точностью до расширения) имена.

Заголовочный файл как правило, содержит объявления, а `.cpp` файл соответствующие им определения.

Заголовочный файл включается во все необходимые файлы с исходным кодом (`.cpp` и `.h`), а имя `.cpp` файлов передаются только компилятору. Эти файлы компилируются как независимые (являются единицами трансляции) и только на последнем этапе компиляции (линковке) объединяются в один исполняемый файл (рис. 1.6). Использовать содержимое `.cpp` файла в других файлах возможно благодаря именно тому, что все объявления (переменных, функций и т.д.) есть в заголовочных файлах.

подробнее о компиляции см. раздел 1.10.3  
Этапы компиляции

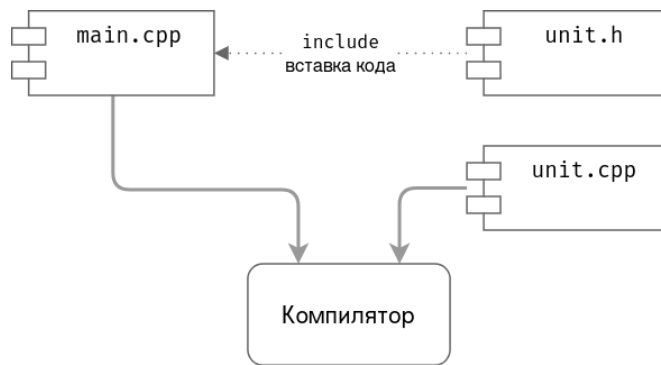


Рис. 1.6. Схема включения и компиляции файлов: код из заголовочных файлов включается в сpp файлы с помощью директивы `include`. Компилятору, передаются только сpp файлы. Причём они компилируются сначала независимо, а потом объединяются

Так как сpp файлы компилируются независимо, то все переменные и функции объявленные в этом файле находятся только в его области видимости.

*unit.h*

```
void print_X();
```

*unit.cpp*

```
// подключать заголовочные файлы стоит именно в сpp файлах (при возможности)
// чтобы избежать нежелательного включения вместе с другими h файлами
#include <iostream>
```

```
int X = 1729;
// функция, использующая переменную X
void print_X(){
    std::cout << X;}

```

*main.cpp*

```
#include <iostream>
#include "unit.h"

int X = 100;           // одноимённая, но _другая_ переменная
int main(){
    std::cout << X;      // 100
    // переменная X из файла unit.cpp -- недоступна по своему имени
    // но можно вывести её значение через функцию print_X() из файла unit.cpp
    print_X();           // 1729
}

```

Однако, стоит избегать использования глобальных переменных.

Чтобы во всех файлах программы была доступна одна и та же переменная X, её нужно объявить (в заголовочном файле) как внешнюю с помощью



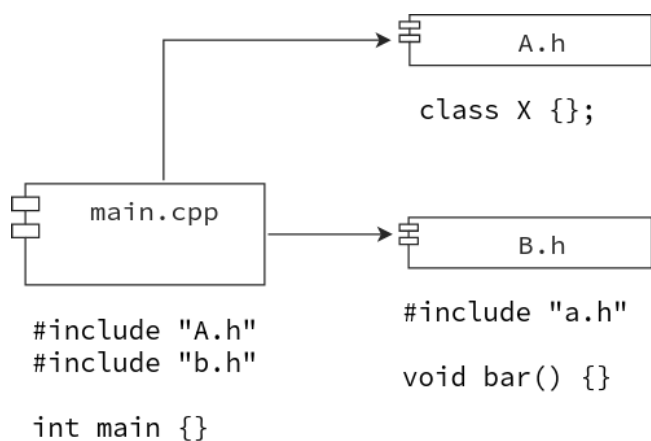
ключевого слова **extern**. **extern int** X; Для компилятора это означает, что переменная X объявлена в одном из cpp файлов.

... Рекомендации по использованию типов данных, переменных, включению файлов и включению пространств имён в заголовочные файлы:

- Типы данных, синонимы типов, заголовки функций объявлять в заголовочных файлах;
- Глобальные переменные объявлять в заголовочных файлах с ключевым словом `extern`, определять в `cpp` файлах;
- `using namespace` и `include` *только* при необходимости в заголовочных файлах, как угодно в `cpp` файлах;
- todo: ...

Подключать заголовочный файл в соответствующий ему `cpp` не обязательно, если последний не *использует* ничего (типы данных, глобальные переменные) из заголовочного файла. Функции могут быть определены в `cpp` файле без подключения заголовочного файла с их объявлениями.

### Защита от повторного подключения



На диаграммах модулей включение показывается стрелкой к включаемым файлам

В результате включения файлов по схеме класс X будет объявлен дважды после обработки директив *#include* препроцессором:

```
// #include "a.h":
class X{};

// #include "b.h":
// #include "a.h":
class X{};

void bar(){};
// ...

int main(){
}
```

Для защиты от повторного объявления при повторном включении заголовочных файлов используются директивы препроцессора.

**Вариант 1.** Директива *#pragma once*.

```
#pragma once

class X{};
// ...
```

pragma – специальная директива для реализации не входящих в стандарт языка C++ возможностей. Изначально *#pragma once* использовалась только в компиляторе MSVC, но потом её поддержка появилась и в других компиляторах.

**Вариант 2** include guards.

```
#ifndef unit_a_h
#define unit_a_h

class X{};
// ...

#endif
```

## 1.9 Модули C++20

**единица трансляции** –

## 1.10 Компиляция программы

### 1.10.1 Компиляция программы из одного файла

Скомпилируем нижеприведённую программу (хранится в файле main.cpp) компилятором G++.

```
#include <iostream>
int main(){
    std::cout << "Hello, World!\n";
    return 0; }
```

```
g++ main.cpp -o hello_world.exe
```

После ключа -o указывается имя исполняемого файла.

Полная поддержка стандартов языка C++ появляется в компиляторах часто спустя несколько месяцев или даже 1-2 года после публикации стандарта. Но отдельные, востребованные нововведения начинают поддерживаться относительно быстро. Иногда нововведения языка могут появиться в компиляторе и раньше принятия стандарта, но это происходит редко. Однако по умолчанию компилятором используется не последний принятый стандарт, а более ранний. Для включения поддержки реализованных возможностей новых стандартов нужно отдельно указывать их название через параметр std:

```
g++ main.cpp -o hello_world.exe -std=C++20
```

### 1.10.2 Макросы препроцессора

[en.cppreference.com/w/cpp/preprocessor/replace](http://en.cppreference.com/w/cpp/preprocessor/replace)

\_\_cplusplus – хранит имя используемого стандарта языка. Может принимать значения: 199711L, 201103L, 201402L, 201703L, 202002L или похожие, в зависимости от компилятора.

Макрос \_\_cplusplus в MSVC: [learn.microsoft.com/ru-ru/cpp/build/reference/zcplusplus?view=msvc-170](https://learn.microsoft.com/ru-ru/cpp/build/reference/zcplusplus?view=msvc-170)

во время работы в оболочке командной строки (например bash или PowerShell) будет полезны команды:

ls – показать файлы в текущей директории

cd <path\_to\_dir>

– изменить текущую папку на указанную (можно указать полный абсолютный или относительный путь);

./my\_program\_name – запустить программу из текущей папки

### 1.10.3 Этапы компиляции

1. **Препроцессинг.** Обработка директив *препроцессора* C++: `include` `define`, `ifdef`, и др. На этом этапе, в том числе, происходит вставка содержимого файлов указанных в директивах `include` (рис. 1.6).
2. Преобразование в Ассемблерный код.
3. Преобразование в машинный код. В результате создаются *объектные файлы* из всех `сpp` файлов переданных компилятору.
4. **Компоновка.** Компоновщик (линкер) используя *таблицу символов* объединяет объектные файлы и файлы статических библиотек в исполняемый файл.

**Таблица символов** – это структура данных, создаваемая самим компилятором и хранящаяся в самих объектных файлах. Таблица символов хранит имена переменных, функций, классов, объектов и т.д., где каждому идентификатору (символу) соотносится его тип, область видимости. Также таблица символов хранит адреса ссылок на данные и процедуры в других объектных файлах. Именно с помощью таблицы символов и хранящихся в них ссылок линкер будет способен в дальнейшем построить связи между данными среди множества других объектных файлов и создать единый исполняемый файл из них.

Детальное описание процесса компиляции:

[en.cppreference.com/w/cpp/language/translation\\_phases](http://en.cppreference.com/w/cpp/language/translation_phases)

### 1.10.4 Компиляция нескольких файлов и статической библиотеки

Предположим, что исходный файл программы разбит на несколько файлов исходного кода:

- `main.cpp` – основной файл, содержит функцию `main`.
- `my_unit1.h`
- `my_unit1.cpp`
- `my_unit2.h`
- `my_unit2.cpp`

Компиляция:

```
g++ main.cpp my_unit1.cpp my_unit2.cpp -o my_prog.exe
```

Отметим, что имена заголовочных файлов не передаются компилятору потому, что их код будет вставлен препроцессором в те места, где из имени указаны в директивах `include` (рис. 1.6).

Для ускорения программы компиляции отдельные сpp файлы можно скомпилировать заранее. Кроме того, исходный код таких файлов уже нельзя будет просмотреть. Эти файлы можно использовать как обычные сpp файлы при компиляции, их код статически (на этапе компиляции) будет включён в исполняемый файл.

todo: пример компиляции библиотеки

[github.com/VetrovSV/OOP/tree/master/examples/example\\_libs/simple\\_lib](https://github.com/VetrovSV/OOP/tree/master/examples/example_libs/simple_lib)

todo: пример компиляции со статической

Динамическая библиотека – это отдельный файл, который не включается в исполняемый файл во время компиляции, а может быть подключен программой в любое время её выполнения. Эти файлы имеют расширение dll в Windows.

### 1.10.5 Параметры компиляции, настройки компиляции в IDE

Некоторые параметры компилятора G++

угловые скобки обозначают обязательные аргумент

- `--version` – показать информацию о версии;
- `-o <my_output_filename>` – имя выходного (исполняемого) файла;
- `-I <include path>` – указывает путь к отдельной (дополнительной) папке с заголовочными файлами;
- `-L <library path>` – указывает путь к отдельной (дополнительной) папке со статическими библиотеками;
- `-l <library>` – указание необходимой статической библиотеки;
- `-O1, O2, O3` – оптимизации кода различного уровня, включая уменьшение размера файла; может замедлять компиляцию;
- `--std=<standard_name>` – указание стандарта языка: `c++17`, `c++20` и др.

todo: параметры компилятора и свойства проекта в MSVC

см. также раздел 2.5 Инструменты разработчика

## 1.11 Параметры командной строки

Полный вариант объявления функции `main` имеет параметры: количество аргументов командной строки (`argc`) и массив из строк (`argv`), который хранит эти аргументы.

```
int main(int argc, char* argv[])
```

Задание значений параметров командной строки для проекта Visual Studio: project, choose Properties, go to the Debugging section – there is a box for "Command Arguments"

todo: ...

## 2 Продвинутые возможности языка

### 2.1 Обработка исключительных ситуаций

**Обработка исключительных ситуаций** (exception handling) – механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Примеры исключительных ситуаций

- Не выполнено предусловие. Например функция ожидает в параметре положительное вещественное число, но передано отрицательное.
- Невозможно создать объект (завершить выполнение конструктора).
- Ошибки типа "индекс вне диапазона".
- Невозможно получить ресурс. Например нет доступа к файлу (файл удалён или не хватает прав доступа).

Исключительные ситуации можно обрабатывать используя коды возврата из функции:

```
/// сортирует массив; возвращает. 1 если a - пустой указатель
int sort_array(float *a, unsigned n){
    if (a == nullptr) // проверка предусловий
        return 1; // если возникла искл.ситуация возвратим 1

    // do sort
    // ...

    return 0; // если всё хорошо, возвращаем 0
}
```

```

//...
int main(){
    float *data = nullptr;

    // ...

    int res = sort_array(data, n);
    // обработка кода возврата:
    if ( res != 0 ){
        cout << "Ошибка!";
    }
    // ...
}

```

Однако использование кодов возврата не всегда возможно или оправдано. Если функция должна возвращать другие данные тогда нужно либо менять возвращаемый тип (чтобы он мог содержать и результат и код возврата), либо предусмотреть другой способ сообщения об исключительной ситуации внутри функции, например через параметр. **todo**: пример. Ещё одна проблема: исключительная ситуация возникает на одно уровне вложенности функции, а возможность её обработки на другом:

```

int foo(float x){
    // тут может возникнуть исключ. ситуация
}
void bar(){
    //...
    foo();
    //...
}
void baz(){
    //...
    bar();
    // обработка искл. ситуации должна быть здесь
    //...
}

```

Рассмотрим специальный механизм обработки исключительных ситуаций. Он состоит из оператора сигнализирования о возникновении исключительной ситуаций (**throw**), специального блока кода («защищённого блока») следящего за возникновением исключений (**try**{ ... }) и блока улавливания сообщений об исключении и их обработки (**catch**( ... ) { ... }).



```

try {
    ...
}
catch ( объявление_переменной-приёмника1 ){
    // обработка исключений, обозначенных типом 1
}
catch ( объявление_переменной-приёмника1 ){
    // обработка исключений, обозначенных типом 2
}
// другие блоки catch
// ...
catch (...){
    // обработка исключений, не подходящих ни под один вышеприведённый тип
}

```

Документация: [en.cppreference.com/w/cpp/language/try\\_catch](http://en.cppreference.com/w/cpp/language/try_catch)

**Пример 1.** Приведём пример программы, которая вычисляет итоговую сумму вклада по формуле сложных процентов. Вынесем вычисления в отдельную функцию. Заранее нельзя быть уверенным в том, что эта функция *всегда* будет вызвана с корректными значениями аргументов. Поэтому предварим вычисления проверкой *предусловий*. Согласно *принципу единственной ответственности* функция не должна решать иных задач, кроме вычисления. Поэтому в ней не стоит сообщать пользователю (например выводом сообщения на экран) о возможной исключительной ситуации.

```

/// возвр. сумму вклада после t начислений процента p, для исходной суммы s
float compound_interest(float s, float p, unsigned t){
    // проверка предусловий:
    if (s <= 0) throw 1;
    if (p <= 0) throw 2;
    if (t == 0) throw 3;
    // вычисления:
    return s * pow(1 + p/100, t);
}

```

Цифрами 1, 2 и 3 обозначены исключительные ситуации, которые могут возникнуть внутри функции.

Обработку этих исключительных ситуаций опишем в той части программы, где понятно как реагировать на исключительную ситуацию. В этом примере, это участок кода где можно вывести сообщение пользователю:

```

#include <iostream>
#include <cmath>

using namespace std;

```

```

int main(){
    float S, S0, percent, tn;
    // ввод данных...
    // ...
    try {
        // защищённый блок кода
        S = compound_interest(S0, percent, tn);
        std::cout << "compound interest =" << S << "\n";
    } catch (int e) {
        // блок обработки исключительных ситуаций
        switch (e) {
            case 1: cout << "Error: S must be greater then zero"; break;
            case 2: cout << "Error: p must be greater then zero"; break;
            case 3: cout << "Error: t must be greater then zero"; break;
        }
    }
    // ...
}

```

Если в функции `compound_interest` возникнет исключительная ситуация, например из-за отрицательного значения аргумента `p`, то её выполнение прервётся в месте вызова **throw**. 2. Выполнение основной программы в секции **try** тоже прервётся, в месте вызова функции `compound_interest`. Выполнение перейдёт в секцию **catch**. В переменную `e` будет записано созданное оператором **throw** значение 2. Наконец, будет выведено соответствующее сообщение на экран.

Если бы оператор **throw** был вызван вне секции **try** или тип брошенного этим оператором значения не соответствует типу переменной в блоке **catch** (например **throw "error";**) то программа завершается аварийно:

```

terminate called after throwing an instance of 'int'
или
terminate called after throwing an instance of 'char*'

```

**Улучшение примера.** Обозначение исключительных ситуаций числами требует от программиста документирования и запоминания их смысла, усложняет понимание программы. Одно из решений – создание перечислений (**enum**) для обозначения таких ситуаций. Но предпочтительнее использовать специальные типы данных [\[en.cppreference.com/w/cpp/error\]](http://en.cppreference.com/w/cpp/error) из стандартной библиотеки, описанный в заголовочном файле (`stdexcept`).

про перечисления см. раздел 1.2.3

Для рассматриваемого примера подходит тип (`invalid_argument`). Он, как и другие аналогичные типы, может содержать текстовое сообщение, поясняющее исключительную ситуацию. Значение этого типа данных создаётся вызовом одноимённо функции.

Приведём пример модифицированной программы:

```
#include <iostream>
#include <stdexcept>
#include <cmath>

using namespace std;

/// возвр. сумму вклада после t начислений процента p, для исходной суммы s
float compound_interest(float s, float p, unsigned t){
    if (s <= 0) throw invalid_argument("S <= 0");
    if (p <= 0) throw invalid_argument("p <= 0");
    if (t == 0) throw invalid_argument("t = 0");
    return s * pow(1 + p/100, t);
}

int main(){
    float S, S0, percent, tn;
    // ВВОД ДАННЫХ...

    try {
        // защищённый блок кода
        S = compound_interest(S0, percent, tn);
        std::cout << "compound interest =" << S << "\n";
    } catch (const invalid_argument &e) {
        // блок обработки исключительных ситуаций
        std::cout << "Error: " << e.what();
    }
    // ...
}
```

Хорошей практикой считается ловить брошенные значения в константные ссылочные переменные, например: `const invalid_argument &e`. Благодаря ссылке, в блоке `catch` вместо создания копии брошенного значения, будет записан только его адрес. Модификатор `const` обеспечит дополнительную строгость программе, запретив непреднамеренное изменение брошенного значения.

Метод (функция вложенная в тип `invalid_argument`) `what` возвращает строковое значение, записанное в значение типа `invalid_argument`.

## Пример обработки исключений при консольном вводе

```
// включить исключения для ввода данных
cin.exceptions(istream::failbit);

float x;
try {
    cin >> x;           // исключение будет брошено, если будет введена строка
                        // которую нельзя преобразовать в вещественное число
    cout << "x = " << x;
} catch (const std::ios_base::failure& fail) {
    cout << fail.what();
}
```

## Область видимости и вызов деструкторов???

...

## Алгоритм выбора подходящего обработчика catch

...

## Вызов throw внутри обработчика исключительных ситуаций throw

...

## Производительность и throw

...

## Спецификатор функции noexcept

... см. также функциональный блок try

## Примеры исключений из стандартной библиотеки??

...

### 2.1.1 Условный оператор, try ... catch, assert

Оператор throw совместно с операторами try...catch стоит использовать только если исключительная ситуация (и соответственно вызов throw) и код её обработки должны находиться на разных уровнях вложенности вызова функций. Т.е. исключительная ситуация может возникнуть внут-

ри функции (или внутри функции, которая вызвана в другой функции и т.д.), а обработка по логике алгоритма возможна только вне этой функции. В остальных случаях для обработки исключительных ситуаций стоит использовать условный оператор.

Например, программа выводящая содержимое текстового файла:

```
// ...
int main(){
    string filename;
    cout << "введите имя файла";
    cin >> filename;
    ifstream f(filename);
    if (f.isopen()){
        // чтение файла, вывод на экран
    }
    else {
        cout << "Невозможно открыть файл: "<<filename; }
    }
```

Добавление в первую ветвь оператора **throw** потребует заключить этот блок в оператор **try** и добавить **catch** сместо ветви **else**. Это только увеличит объём кода и затруднит его чтение.

Оператор `assert` тоже используется для проверок. Но если **try ... catch** используется для реагирования на исключительные ситуации во время работы уже соданной программы, которые могут возникнуть из-за некорректных данных или недоступности ресурсов. То оператор `assert` нужен для автоматических тестов во время разработки программы, т.е. проверки корректности кода, а не данных.

## 2.2 Функции

### 2.2.1 Статические локальные переменные

...

### 2.2.2 Переменное число параметров функции (variadic arguments)

- Документация:  
[en.cppreference.com/w/cpp/language/variadic\\_arguments](http://en.cppreference.com/w/cpp/language/variadic_arguments)
- Описание и примеры:  
[ravesli.com/urok-111-ellipsis-pochemu-ego-ne-sleduet-ispolzovat/](http://ravesli.com/urok-111-ellipsis-pochemu-ego-ne-sleduet-ispolzovat/)

### 2.2.3 Указатель на функцию

Общий синтаксис описания типа указателя на функцию:

```
return_type (*) (arg1_type, arg2_type, ... );
```

Например, функциональный тип указывающий на любую функцию, которая принимает один аргумент типа `int` и возвращает значение типа `float`:

```
float (*) (int);
```

Для упрощения записи подобных типов создают синонимы

```
using FuncIntFloat = float (*) (int);  
// FuncIntFloat -- синоним
```

Эти функции соответствуют типу `FuncIntFloat`:

```
float sqrt(int x){return pow(x,0.5);}

float foo(int x){return x*x * 22.0/7; }

// адреса функций можно записать в переменные типа FuncIntFloat
FuncIntFloat sq_root = &sqrt;
FuncIntFloat bar = &foo;
```

Функциональный тип может быть аргументом другой функции

```
/// выводит элементы массива, преобразуя их функцией f
void array_apply_n_print(int *arr, unsigned n, FuncIntFloat f){
    for (unsigned i=0; i<n; i++) {
        cout << f(arr[i]) << " ";
    }
}

int main()
{
    unsigned N = 8;
    int* a = new int[N] {1,2,3,4,5,6,7,8};

    cout << endl;
    array_apply_n_print(a, N, sqrt);
    cout << endl;
    array_apply_n_print(a, N, foo);

    cout << endl;
    // вместо адреса созданной функции, можно передать анонимную функцию
```

```

array_apply_n_print(a, N, [](int x)->float{return x;} );
// функция [](int x)->float{return x;} возвращает свой аргумент неизменным
cout << endl;
array_apply_n_print(a, N, [](int x)->float{return x*x;} );
// функция [](int x)->float{return x*x;} возвращает квадрат числа
return 0;
}

```

Вывод программы:

```

1 1.41421 1.73205 2 2.23607 2.44949 2.64575 2.82843
0.318182 1.27273 2.86364 5.09091 7.95455 11.4545 15.5909 20.3636
1 2 3 4 5 6 7 8
1 4 9 16 25 36 49 64

```

## 2.2.4 Перегрузка операторов (operator overloading)

Оператор – это функция со специальным, символьным именем.

Операторы доступные для перегрузки: + - \*

Для сложения комплексных чисел описанных структурой

```

struct ComplexNumber{
    double re, im;
};

```

можно создать отдельную функцию:

```

/// возвращает результат сложения комплексных чисел
ComplexNumber complex_plus(const ComplexNumber& a, const ComplexNumber& b){
    ComplexNumber c;
    c.re = a.re + b.re;
    c.im = a.im + b.im;
    return c; }

```

Тогда вычисления с этой функцией могут выглядеть так:

```

ComplexNumber c1{1,2}, c2{3,-6};
ComplexNumber s1 = complex_plus(c1, c2);

```

Однако более естественной была бы запись сложения с оператором +. Но оператор сложения не имеет реализаций для работы с нестандартным типом ComplexNumber.

Создадим новую функцию вместо старой – оператор сложения, заменив только название старой функции на обозначение оператора сложения:

**operator** +.

```
ComplexNumber operator +(const ComplexNumber& a, const ComplexNumber& b){  
    ComplexNumber c;  
    c.re = a.re + b.re;  
    c.im = a.im + b.im;  
    return c; }
```

Первый параметр этой функции – левый операнд, второй – правый операнд. Тогда вычисления можно записывать так:

```
ComplexNumber c1{1,2}, c2{3,-6};  
ComplexNumber s1 = c1 + c2;
```

Таким образом, добавлена новая реализация оператора сложения, помимо существующих реализаций для стандартных типов вроде float или int. Вызов оператора сложения отличается от аналогичной функции только способом записи операндов при вызове.

## 2.2.5 Анонимные функции

```
[ захват ] ( параметры ) attr -> возвращаемый_тип { тело }
```

**Захват** - глобальные переменные используемые функцией (по умолчанию не доступны),

**параметры** - параметры функции; описываются как для любой функции,

**mutable** - указывается, если нужно поменять захваченные переменные,

**исключения** - которые может генерировать функция,

**атрибуты** - те же что и для обычных функций.

Возведение аргумента в квадрат

```
[](auto x) {return x*x;}
```

Сумма двух аргументов

```
[](auto x, auto y) {return x + y;}
```

Возведение аргумента в квадрат



```
[](auto x) {return x*x;}
```

Сумма двух аргументов

```
[](auto x, auto y) {return x + y;}
```

Вывод в консоль числа и его квадрата

```
[](float x) {cout << x << " " << x*x << endl;}
```

Тело лямбда-функции описывается также как и обычной функции

```
[](int x) { if (x % 2) cout << "н"; else cout << "ч"; }}
```

Использование захвата.

= - захватить все переменные.

& - захватить переменную по ссылке.

Чтобы изменять переменную захваченную по ссылке нужно добавить **mutable** к определению функции.

```
float k = 1.2;  
float t = 20;
```

```
[k](float x) {return k*x;}
```

```
[k,&c](float x) mutable {if (k*x > 0) c = 0; else c=k*x;}
```

Когда использовать лямбда функции?

Когда не требуется объявлять функцию заранее.

Функция очень короткая.

Функция нужна один раз.

Функцию лучше всего описать там, где она должна использоваться.

## Примеры использования decltype и auto для указателей на функции

...

### std::function

Тип `std::function` предоставляют более удобный вариант передачи функций в качестве аргументов, в том числе анонимных функций с захватом внешних переменных.

```
/// перебирает значения элементов массива arr размером n,
/// для каждого элемента вызывает функцию f
void apply_to_array(int * arr, unsigned n, std::function<void(int&)> f){
    // std::function<void(int&)> -- указатель на функцию, возвращающую void,
    // принимающую int по ссылке
    for (unsigned i = 0; i < n; ++i)
        f( arr[i] );
}

int a[] = {1,2,3,4,5,6};

apply_to_array(a, 6, [](int &x){cout << x << " ";});
cout << "\n";

apply_to_array(a, 6, [](int &x){cout << x*x << " ";});
cout << "\n";

apply_to_array(a, 6, [](int &x){x = x+1;});

apply_to_array(a, 6, [](int &x){cout << x << " ";});
cout << "\n";

// взаимодействие с внешними переменными:
stack<int> S;
apply_to_array(a, 6, [&S](int &x){S.push(x);});
// [&S] - захват, позволяет использовать внешние переменные;
// в данном случае функции доступна ссылка на S
```

[en.cppreference.com/w/cpp/utility/functional/function](http://en.cppreference.com/w/cpp/utility/functional/function)

### 2.2.6 Автоматическое тестирование

Программист должен быть уверен в корректности программы. Тестирование всех функций – один из способов повысить качество кода. Тестирование большинства функций можно автоматизировать. Выполнять эти функции (на этапе компиляции или на этапе запуска программы) с заранее заданными параметрами и сравнивать их результат с заранее вычисленным и гарантированно правильным результатом.

Для примера создадим тесты для функции вычисления среднеквадратичного отклонения выборочных данных:  $std = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - m)^2}$ , где  $m$  – среднее значение выборки  $X$ .

Заранее подготовим тестовые данные:

$x_1 = (1, 2, 3, 4, 5, 6), std = 1.707825127659933$

$x_2 = (216.68167942, -472.1980655, -155.84417437, 641.71416684, -330.48466343,$

$426.13143811, -565.02650103, 730.202725, -655.47498993, -336.26766006),$

$std = 486.51128339480533$

$x_3 = (42), std = 0$

$x_4 = (100, 200), std = 50$

Тестируемая функция:

```
/// возвращает с.к.о значений из массива a размером n
double array_std(double *a, unsigned long n){
    double s = 0.0;
    double m = 0.0;

    // вычисление среднего значения
    for (unsigned long i = 0; i < n; i++) m += a[i];
    m /= n;

    for (unsigned long i = 0; i < n; i++) s += pow(a[i] - m, 2);
    s = sqrt(s/n);

    return s; }
```

Из-за погрешностей в вычислениях, результат функции и ожидаемый результат могут незначительно отличаться. Например в десятой значащей цифре. Поэтому прямое сравнение этих двух вещественных оператором `==` скорее всего даст отрицательный результат. Будем считать, что функция выдала правильный результат если он по модулю отличается от ожидаемого не более чем на заранее заданную величину  $\epsilon$ . Эту величину будем считать допустимой ошибкой и положим равной  $10^{-8}$ .

Тогда сравнение будет иметь вид

```
const double EPS = 10^{-8};
abs( f - 1.707825127659933 ) < EPS
```

Для тестирования вместо обычного условного оператора используем макрос `assert`. Если переданное в макрос логическое выражение ложно, то он аварийно завершает программу и выводит диагностическое сообщение. Например:

```
#include <cassert>
// ...

assert( 2+2 == 5);
```

Сообщение в консоли об ложном условии в строке 5 файла main.cpp

```
main: main.cpp:5: int main(): Assertion `2+2 == 5' failed.  
Aborted (core dumped)
```

Пример 1. Тестирования функции:

```
#include <iostream>  
#include <cassert>  
#include <math.h>  
#include "stats.h" // файл с тестируемой функцией array_std  
  
using namespace std;  
  
int main(){  
    const double EPS = 10E-8; // допустимая ошибка  
    // тестовые данные:  
    double *x1 = new double[6] {1,2,3,4,5,6}; // std = 1.707825127659933  
    double *x2 = new double[10] {216.68167942, -472.1980655, -155.84417437,  
                                   641.71416684, -330.48466343, 426.13143811,  
                                   -565.02650103, 730.202725, -655.47498993,  
                                   -336.26766006}; // std = 486.51128339480533  
    double *x3 = new double[10] {42}; // std = 0  
    double *x4 = new double[10] {100, 200}; // std = 50  
  
    // тестирование:  
    assert( abs(array_std(x1,6) - 1.707825127659933) < EPS );  
    assert( abs(array_std(x2,10) - 486.51128339480533) < EPS );  
    assert( abs(array_std(x3,1)) < EPS );  
    assert( abs(array_std(x4,2) - 50.0) < EPS );  
  
    // если все тестовые случаи завершились успешно, то:  
    cout << "Test array_std OK\n";  
}
```

Тестирование должно выполняться автоматически при каждом запуске программы, пока она находится в разработке, и не должен требовать от программиста или пользователя дополнительных действий. Вызов тестовых функций должен предварять код основной логики программы. Ведь если функции в программе работают неправильно, то скорее всего не имеет смысла выполнять программу дальше. Тестовые данные должны быть как можно разнообразнее, в том числе затрагивать **крайние случаи**. Имеет смысл вынести тестирующий код в отдельную функцию, отдельный файл исходного кода.

## Пример 2. Тестирования функции:

```
// main.cpp:
#include <iostream>
#include <cassert>
#include <math.h>

#include "stats.h"
#include "test.h"    // файл с функцией тестирования

using namespace std;

int main(){
    const double EPS = 10E-8;    // допустимая ошибка
    // тестовые данные:
    double *x1 = new double[6]   {1,2,3,4,5,6};    // std = 1.707825127659933
    double *x2 = new double[10]  {216.68167942, -472.1980655, -155.84417437,
                                   641.71416684, -330.48466343, 426.13143811,
                                   -565.02650103, 730.202725, -655.47498993,
                                   -336.26766006};    // std = 486.51128339480533
    double *x3 = new double[10]  {42};              // std = 0
    double *x4 = new double[10]  {100, 200};         // std = 50

    // тестирование:
    assert( abs(array_std(x1,6) - 1.707825127659933) < EPS );
    assert( abs(array_std(x2,10) - 486.51128339480533) < EPS );
    assert( abs(array_std(x3,1)) < EPS );
    assert( abs(array_std(x4,2) - 50.0) < EPS );

    // если все тестовые случаи завершились успешно, то:
    cout << "Test array_std OK\n";
}
```

## Итоговый пример тестирования функции:

```
// test.h:
#include <assert.h>
#include <iostream>
#include "stats.h"

/// тестирует функцию array_std
void test_array_std(){
    const double EPS = 10E-8;    // допустимая ошибка
    // тестовые данные:
    double *x1 = new double[6]   {1,2,3,4,5,6};    // std = 1.707825127659933
    double *x2 = new double[10]  {216.68167942, -472.1980655, -155.84417437,
                                   641.71416684, -330.48466343, 426.13143811,
                                   -565.02650103, 730.202725, -655.47498993,
                                   -336.26766006};    // std = 486.51128339480533
    double *x3 = new double[10]  {42};              // std = 0
    double *x4 = new double[10]  {100, 200};         // std = 50

    static_assert( abs(array_std(x1,6) - 1.707825127659933) < EPS );
    assert( abs(array_std(x2,10) - 486.51128339480533) < EPS );
    assert( abs(array_std(x3,1)) < EPS );
}
```

```

    assert( abs(array_std(x4,2) - 50.0) < EPS );
    std::cout << "Test array_std OK\n";
}

```

```

// stats.h:
#pragma once

```

```

/// возвращает с.к.о значений из массива a размером n
double array_std(double *a, unsigned long n);

```

```

// stats.cpp:
#include <math.h>

```

```

/// возвращает с.к.о значений из массива a размером n
double array_std(double *a, unsigned long n){
    double s = 0.0;
    double m = 0.0;

    // вычисление среднего значения
    for (unsigned long i = 0; i < n; i++)
        m += a[i];
    m /= n;

    for (unsigned long i = 0; i < n; i++)
        s += pow(a[i] - m,2);
    s = sqrt(s/n);

    return s;
}

```

```

// main.cpp:
#include <assert.h>
#include <math.h>

```

```

#include "test.h"
#include "stats.h"

```

```

int main(){
    // вызов тестирующей функции
    test_array_std();

    // основной алгоритм программы:
    // ...
}

```

См. также static\_assert, google test, тестовый проект в VS

См. также **модульное тестирование** в разделе 4.5.

todo: выключение assert

todo: assert с выводом сообщения (и оператор ,)

todo: Test Driven Development (TDD)

## Ссылки

1. О тестировании: [github.com/VetrovSV/OOP/blob/master/unit\\_test/unit\\_test.md](https://github.com/VetrovSV/OOP/blob/master/unit_test/unit_test.md)
2. Совершенный код (2-е издание, 2005 г.) Стив Макконнел

## 2.3 Шаблонные функции

Документация: [en.cppreference.com/w/cpp/language/function\\_template](http://en.cppreference.com/w/cpp/language/function_template)

*Обобщённое программирование* – todo

*Статический полиморфизм* – todo

Синтаксис *определения* шаблонной функции:

```
template < parameter_list >
function_declaration
```

описание шаблонных типов может быть на одной строке с заголовком функции, но это ухудшает читаемость

parameter\_list – шаблонные параметры;

При описании шаблонного типа возможно указывать:

**typename** – todo

**class** – todo

Пример объявления шаблонной функции:

```
// шаблонная функция
template < typename Element >
Element sum_array(Element* arr, unsigned n){
    Element sum = 0;
    for (unsigned i = 0; i < n; ++i)
        sum += arr[i];
}
```

часть стандартной библиотеки STD – Standard Template Library содержит большое количество шаблонных функций и классов

Пример вызова шаблонной функции:

```
int main() {
    unsigned N1 = 6;
    int *arr1 = new int[N1] {42, 43, 44, 45, 46, 47};
}
```

```

unsigned N2 = 4;
double *arr2 = new double[N2] {5./7, 5./11, 5./13, 5./17};

// компилятор создаст вариант функции с Element = int
int S1 = sum_array(arr1, N1);

// компилятор создаст вариант функции с Element = double
cout << sum_array(arr2, N2) << "\n";

// тип для шаблона можно указывать явно:
cout << sum_array<double>(arr2, N2);

unsigned N3 = 128;
Point *arr3 = new Point[N3];

// ошибки: невозможно инициализировать значение типа Point нулём
// оператор + не определён для типа Point
// невозможно создать вариант функции с Element = Point:
Point S3 = sum_array(arr3, N3);
}

```

Если шаблонный тип не указан явно, то определяется по типам аргументов функции.

Шаблонный тип данных несовместимый с функцией создаёт ошибку компиляции:

```

struct Point{
    float x;
    float y;
};

int main() {
    unsigned N3 = 128;
    Point *arr3 = new Point[N3];

    // ошибки: невозможно инициализировать значение типа Point нулём
    // оператор + не определён для типа Point
    // невозможно создать вариант функции с Element = Point
    Point S3 = sum_array(arr3, N3);
}

```

Если для данного типа компилятор не может реализовать функцию, то можно определить **специализированную шаблонную функцию**:

```

template <>
Point sum_array(Point* arr, unsigned n){
    Point sum {0,0};
    for (unsigned i = 0; i < n; ++i) {
        sum.x = sum.x + arr[i].x;
    }
}

```

проблему перегрузки функции можно решить ещё и определив конструктор преобразования для типа Point, чтобы операция Point sum = 0; стала возможной (о конструкторах преобразования см. раздел 4.3.1) и создать реализацию оператора



```

        sum.y = sum.y + arr[i].y;
    }
    return sum; }

```

Компилятор просматривает все вызовы шаблонной функции, и если код функции позволяет подставить указанный в шаблоне тип данных, то создаётся функция с данным типом. Если функций больше одной, то они будут перегружены.

Такая генерация функций возможна только если функция определена в той же единице трансляции (срр файле), где происходит вызов. Связывание же скомпилированных файлов происходит на последнем этапе компиляции, до этого компилятор обрабатывает срр файлы и библиотеки практически независимо.

Таким образом Шаблонные функции могут быть объявлены только в том же месте, где определены, т.е. в заголовочных файлах, которые, в конце концов, включаются в срр файлы где вызываются.

Если шаблонная функции не вызвана, то и не создаётся ни одного варианта этой функции.

Возможно описать шаблонную функцию с несколькими шаблонными типами, в том числе вложенными.

См. также раздел 4.4 Шаблонные классы.

## 2.4 Ассемблерные вставки

Общий синтаксис вставки ассемблерного кода для компилятора MSVC:

```

__asm {
    asm_code
};

```

Пример:

```

#include <iostream>

int main()
{
    int x = 42;
    short y = 100;

    __asm
    {

```

пример приведён для проекта MSVC, который компилируется для процессоров x86; используется Intel-синтаксис ассемблерного кода (см. также AT&T синтаксис)

```

        mov x, 10
        mov ax, 99
        mov y, ax
    }

    std::cout << x << "\n";    // 10
    std::cout << y << "\n";    // 99
}

```

## 2.5 Инструменты разработчика

### 2.5.1 Обзор

В разделе 1.10 “Компиляция программы” были приведены примеры команд компиляции для простых программ, состоящих из нескольких файлов исходного кода. Более сложные программы обычно полагаются на внешние библиотеки, наборы дополнительных файлов (файлы с данными, файлы настроек, изображения и т.д.). Иногда такие программы компилируют частями, разбивая их на отдельные модули. Таким образом, процесс компиляции может включать в себя несколько запусков компилятора с различными файлами и настройками, с учётом расположения ранее скомпилированных частей программы, запуск тестов, очистку папок от временных файлов, копирование уже готовой программы и её файлов в отдельный, от файлов исходного кода, каталог.

**Система автоматизации сборки** – набор программ, призванных упростить компиляцию кода, с учётом параметров компилятора, зависимостей кода (других библиотек), выполнения модульных тестов, очистки проекта от временных файлов и т.п. Система сборки полагается на специальные конфигурационные файлы, которые описывают состав проекта и процесс его компиляции. Большинство современных IDE при компиляции программы не вызывают компилятор напрямую, а обращаются к системе сборки.

**CMake** – система сборки, которая позволяет автоматизировать компиляцию программ (сконфигурировать команду компиляции), выполнение тестов, развёртывание программы и т.п. (см. пример: [en.wikipedia.org/wiki/CMakeHello\\_world](https://en.wikipedia.org/wiki/CMakeHello_world))

Настройки сборки описываются в файле `CMakeLists.txt`. Например:

```

project(my_project)                                # Присваиваем имя проекту

# Цель сборки -- исполняемый файл
add_executable(
    ${PROJECT_NAME}                                # Имя исполн. файла = имя проекта
    main.cpp                                         # Список файлов исходного кода
)

```

Пример: [github.com/VetrovSV/OOP/tree/master/examples/CMake](https://github.com/VetrovSV/OOP/tree/master/examples/CMake)

Conan – менеджер зависимостей для C++. Программа позволяет скачивать указанные библиотеки; [ [docs.conan.io/en/latest/introduction.html](https://docs.conan.io/en/latest/introduction.html) ]

NuGet – менеджер пакетов

cling – интерпретатор для C++

git – система контроля версий.

diff – программа для сравнения текстовых файлов; может применяться для сравнения файлов исходного кода.

**clang-format** – консольная программа для форматирования исходного кода согласно стилю оформления. Пример: `clang-format -i <filename>`

markdown – облегчённый стандарт разметки текстовых файлов. Применяется для оформления простых тестовых документов, в частности на GitHub в readme файлах.

Онлайн-IDE??

Статические анализаторы?

Профилирование??

bash + GNU tools??

## 2.5.2 Visual Studio


todo:

Структура проекта и решения

Профили компиляции, запуск с отладкой vs без отладки

Настройка параметров компиляции

Отладка, условные точки останова

Переименование (  + ,  +  )

Интеграция с git















Интеграция автоматических тестов (см. также 4.5)

Профилирование

Live Share

TODO

Некоторые операции и горячие клавиши:

- Автоматическое форматирование кода ( , , ,  )
- Меню быстрых операций над кодом ( ,  )
- Сборка ( ,  )
- Закомментировать выделение ( , , ,  )
- Перейти к предыдущей позиции курсора ( ,  )

Подробнее: [learn.microsoft.com/en-us/visualstudio/ide/default-keyboard-shortcuts-in-visual-studio?view=vs-2022](https://learn.microsoft.com/en-us/visualstudio/ide/default-keyboard-shortcuts-in-visual-studio?view=vs-2022)

## 3 Стандартная библиотека

**Стандартная библиотека** C++ содержит многое, что необходимо для хранения и обработки данных (динамический массив, список, и т.д.), для работы с файлами, сетью, потоками и др. Модули для создания приложений с GUI в состав библиотеки не входят.

Набор классов и функций ...

- для хранения данных - контейнеры (строки, список, динамический массив, словарь, ...)
- для обработки данных - алгоритмы (сортировка, поиск, поэлементная обработка, ...)
- для ввода и вывода на экран
- для файлов и файловой системы
- для параллельного программирования
- ...

### 3.1 string

```
#include <string>
string s = "Hello!";
string s2("Hello");
string s1 = string("Hello");
s[ 0 ];
s.at(0);
s.insert(0,"xx");
cout << s << endl;
s.size();
s.length();
s.empty();
s.clear();
// преобразование в массив символов
const char *ss = s.c_str();
// конкатенация
```

```
string s3 = s1 + s2;
s1 += s2;
```

## 3.2 string stream

`sstream` – класс для работы со строками, через интерфейс потоков. Принцип работы с классом такой же как и в других потоках, для вывода (`cout`) и файловых (`fstream`), но все записанные данные помещаются в строку (`string`). Возможно преобразование числовых типов в строку.

```
#include <sstream>
#include <iomanip>          // для setprecision

std::ostringstream s;
int n = 42;
float x = 123.456789;
s << "n = " << n << "; x = " << x;

// преобразование в строку (.str()) и вывод на экран
std::cout << s.str() << std::endl;

s.str(""); // очистка потока
// настройка вывода: не экспоненциальный формат, 2 знака после запятой
s << std::fixed << std::setprecision(2);
s << "x = " << x << "; pi " << 333./106;
std::cout << s.str();
```

## 3.3 Умные указатели

Приведём пример функции с утечкой памяти:

```
void print_random_word_leak(unsigned len){
    // выделение памяти в куче (динамической памяти)
    std::string *s = new std::string ();

    // составление строки из случайных букв английского алфавита
    for (int i = 1; i < len-1; ++i) {
        // добавление новой буквы в строку
        s->push_back( (char)(97 + rand()%26) ); // 97 - код буквы 'a'
    }

    std::cout << *s;
    // утечка памяти: память по адресу s не освобождена, указатель s будет утерен после заверш
}

int main(){
```

```

    srand(time(0));

    print_random_word_leak(9);
}

```

Утечка памяти может происходить по многим причинам, не только из-за забытого оператора delete. Например в функции print\_random\_word может возникнуть исключение во время добавления символа в строку, если размер строки превышает максимально возможный. Тогда даже если в конце функции будет предусмотрено освобождение памяти по указателю s, оно не будет выполнено из-за брошенного исключения.

Такую проблему решают специальные типы данных – умные указатели. Это шаблонный класс, который автоматически освобождает память объекта, которым владеет. Они представляют собой обёртку вокруг классических указателей.

```
#include <memory>
```

```
std::unique_ptr
std::make_unique
```

Создадим локальную переменную типа unique\_ptr которая перед своим уничтожением автоматически освободит и ту память, которую её передали для владения:

```

void print_random_word(unsigned len){
    // умный указатель. шаблонный тип -- тип, на который указатель указывает
    unique_ptr<string> s ( new string () );

    // составление строки из случайных букв английского алфавита
    for (int i = 1; i < len-1; ++i) {
        // добавление новой буквы в строку
        s->push_back( (char)(97 + rand()%26) ); // 97 - код буквы 'a'
        // -> -- перегруженный оператор, который позволяет прозрачно обращаться
        // к членам класса, указатель на который хранится внутри unique_ptr
    }

    // * -- перегруженный оператор, который разыменовывает обёрнутый указатель
    std::cout << *s;
    // аналогично:
    // std::cout << *s.get();
    // s.get() вернёт string *

    // s -- локальная переменная, уничтожится при завершении функции
    // перед уничтожением s будет вызван её деструктор
    // который освободит память по указателю на строку
}

```

Если использовать специальную функцию `make_uniq` то можно не делая и явного выделения памяти

```
void print_random_word2(unsigned len){
    // make_unique самостоятельно вызывает оператор new
    unique_ptr s = make_unique<string>( string () );

    // составление строки из случайных букв английского алфавита
    for (int i = 1; i < len-1; ++i) {
        // добавление новой буквы в строку
        s->push_back( (char)(97 + rand()%26) );    // 97 - код буквы 'a'
        // -> -- перегруженный оператор, который позволяет прозрачно обращаться
        // к членам класса
    }

    // * -- перегруженный оператор, который разыменовывает обёрнутый указатель
    std::cout << *s;
    // аналогично:
    // std::cout << *s.get();
    // s.get() вернёт string *

    // s -- локальная переменная, уничтожится при завершении функции
    // перед уничтожением s будет вызван её деструктор
    // который освободит память по указателю на строку
}
```

todo: отличия `uniq_ptr` от `shared_ptr`;

todo: RAII

## 3.4 Контейнеры

### 3.4.1 Общее описание

Некоторые контейнеры:

- `vector` – динамический массив;
- `list` – динамический массив;
- `stack` – стек;
- `queue`, `deque` – очередь, двусторонняя очередь;
- `set` – множество;
- `map` – словарь (ассоциативный массив);
- ...



Container	Overhead	Iterators	Insert	Erase	Find
list	8	Bidirectional	amortized constant	amortized constant	N
deque	12	Random	amortized constant at begin or end; else N/2	amortized constant at begin or end; else N	N
vector	0	Random	amortized constant at end; else N	amortized constant at end; else N	N
set	12	Bidirectional	log N	log N	log N
multiset	12	Bidirectional	log N	d log (N+d)	log N
map	16	Bidirectional	log N	log N	log N
multimap	16	Bidirectional	log N	d log (N+d)	log N

Рис. 3.1. Overhead – дополнительное количество байт для хранения одного элемента внутри контейнера. Например для списков требуется дополнительная память для хранения указателей. Столбец Iteratos обозначает способ доступа к элементам. Bidirectional – для доступа к произвольному элементу требуется пройти от начала или конца контейнера, Random – произвольный доступ (по индексу). Эффективность операций вставки (insert), удаления (erase) и поиска обозначена функцией числа элементов. Например  $\log N$  означает, что операция занимает время пропорциональное логарифму числа элементов  $N$  контейнера. Реализации некоторых операций оптимизирована на случай частого вызова. Например время добавления нового элемента в конец динамического массива – константа, т.е. почти не зависит от количества элементов массива. Это происходит из-за того, что vector при добавлении нового элемента резервирует дополнительное место в массиве, на случай если будут добавлены ещё элементы. Источник

### 3.4.2 vector

```
#include <iostream>
#include <vector>

using namespace std;

// создание синонима для типа vector<int>
using vector_int = vector<int>;
// vector -- класс-обёртка для динамического массива
// vector -- шаблонный класс, поэтому поддерживает задание типа
// для вложенных в него значений (здесь это элементы массива).
// Тип вложенных значений указывается внутри угловых скобок
// < > при объявлении переменной типа vector

/// вывод динамического массива
void print_vector(const vector_int &v ){
    // вектор передаётся по ссылке чтобы избежать лишнего копирования
```

```

// т.к. эта функция не должна менять вектор, то делаем формальный параметр константой
// фактический параметр не обязательно должен быть константой
for (int i = 0; i < v.size(); ++i)
    cout << v[i] << " ";

int main(int argc, char const *argv[]){

    vector<int> arr;           // динамический массив (пока пустой)
    arr.resize( 100 );        // изменение размера.
    unsigned n = arr.size();   // -> размер
    // обращение к элементам
    arr[0] = 42;
    print_vector( arr );
    arr.clear();              // освобождение памяти
    // функция clear вызывается автоматически при уничтожении переменной

    // матрица - вектор из векторов
    vector< vector<int> > matr;
    // выделение памяти под 10 элементов (с типом vector<int> )
    matr.resize(10);
    // выделение памяти под строки матрицы
    // 25 столбцов или элементов в каждой строке
    for (int i = 0; i < matr.size(); ++i)
        matr[i].resize(25);

    return 0;
}

```

## 3.5 Алгоритмы

Некоторые функции, реализующие часто используемые алгоритмы:

- find, find\_if
- max, min
- sort
- is\_sorted

## 3.6 Файловые потоки

```
#include <fstream>

using namespace std;
```

Для работы с файлами в C++ используется принцип схожий с потоками ввода (cout) и вывода (cin).

Основные типы (классы) для работы с файловыми потоками.

- ifstream – input filestream, чтение файлов
- ofstream – output filestream, запись в файл
- fstream – output filestream, чтение и запись в файлы.

### Запись в файл

```
#include <fstream>
using namespace std;
...
// создать объект f для записи в файл, открыть файл
// если файл не существует, то будет создан, если существует, то будет перезаписан
ofstream f("myfile.txt");

if ( f.is_open() ) {    // проверка, удалось ли открыть файл
    // запись в файл:
    // здесь все данные будут записаны слитно. так лучше не делать
    f << "qwerty";
    f << 123;
    f << 3.14;
    f << "\n"; // записать символ перехода на новую строку
    f << 42.5;
    f.close(); // закрытие файла, в т.ч. запись файлового буфера на экран
}
```

Содержимое созданного файла:

```
qwerty1233.14
42.5
```

[https://en.cppreference.com/w/cpp/io/basic\\_ofstream](https://en.cppreference.com/w/cpp/io/basic_ofstream)

Запись `ofstream f("myfile.txt");` аналогична записи `ofstream f = ofstream("myfile.txt");`.  
Здесь инициализируется переменная `f` и открывается файл для записи.

Можно создавать переменную для работы с файлами не открывая файл сразу, а для открытия использовать метод `open`:

```
ofstream f;           // создать объект f для записи в файл
f.open("myfile.txt"); // открытие файла
```

Функция открытия (open) и функция инициализации и открытия могут принимать во втором параметре режим работы с файлом. Константы, определяющие эти режимы определены в пространстве имён `std::ios_base`.

- `app` — открыть файл для дополнения;
- `binary` — открыть файл для работы с ним как с бинарным файлом, по умолчанию файлы открываются как текстовые;
- `in` — открыть для записи (по умолчанию для `ifstream`);
- `out` — открыть для чтения (по умолчанию для `ofstream`);
- `trunc` — перезаписать файл при открытии (по умолчанию для `ofstream`);
- `ate` — перейти в конец файла после открытия.

Непротиворечивые режимы можно комбинировать между собой с помощью логического оператора или (`|`). Например:

```
/// открыть файл example.bin для записи, в режиме дополнения, работать как с бинарным
myfile.open("example.bin", ios::out | ios::app | ios::binary);
```

[en.cppreference.com/w/cpp/io/ios\\_base/openmode](https://en.cppreference.com/w/cpp/io/ios_base/openmode)

## Чтение из файла

```
#include <fstream>
using namespace std;

// создать экземпляр класса ifstream (для чтения файлов)
ifstream f1;
// открыть текстовый файл
f1.open("myfile");
if (f1.is_open()){
    string s;
    f1 >> s; // s = "qwerty1233.14"
    ...
    f1 >> s; // s = "42.5"
    float number = stof(s); // строка -> число
    f1.close();
}
```

[https://en.cppreference.com/w/cpp/io/basic\\_ifstream](https://en.cppreference.com/w/cpp/io/basic_ifstream)

## Построчное чтение файла

```

#include <fstream>
using namespace std;

ifstream f;
f.open(filename);
if (f.is_open()){
    string buf;
    /// getline вернёт значение, преобразуемое в false если не осталось строк для чтения
    while ( getline(f,buf) ){
        cout << buf << endl;
    }
f.close();}

```

getline проигнорирует последнюю пустую строку в файле, но прочитает пустую строку в начале или середине.

### Перемещение по файлу при чтении

```

ifstream f;
f.open(filename);
if (f.is_open()){
    string buf;
    /// первая строка будет прочитана два раза
    getline(f,buf);
    cout << "buf = " << buf << endl;
    f.clear();
    f.seekg(0); /// сбросить бит конца файла, чтобы переместить указатель в файле
    f.tellg(); /// = 0; вернёт позицию в файле
    /// некоторые символы, например из кириллицы,
    /// занимают больше одного байта
    cout << "buf = " << buf << endl;

    while (getline(f,buf)) ; /// чтение файла до конца
    /// после попытки чтения строки, когда конец файла уже достигнут,
    /// в файловой переменной f будет установлен флаг fail
    /// seekg с этим флагом не работает
    /// поэтому нужно очистить все флаги перед перемещением
    f.clear();
    f.seekg(0); /// в начало
    cout << "buf = " << buf << endl; }

```

Бинарные файлы: [github.com/VetrovSV/OOP/blob/master/2021-fall/bin-files.md](https://github.com/VetrovSV/OOP/blob/master/2021-fall/bin-files.md)

## 3.7 Регулярные выражения (regex)

Регулярные выражения (regular expressions, RegEx) – формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов.

Для поиска используется строка-образец (pattern, «шаблон», «маска»), состоящая из символов и метасимволов и задающая правило поиска.

Веб-приложение для проверки регулярных выражений: [regex101.com](http://regex101.com)

Шпаргалка по регулярным выражениям: [exlab.net/files/tools/sheets/regexp/regexp.pdf](http://exlab.net/files/tools/sheets/regexp/regexp.pdf)

**bool** `regex_match(строка, regex)` – функция поиска соответствий в строке регулярному выражению `regex`; возвращает **true**, если соответствие найдено.

Документация по модулю `regex`:

<https://en.cppreference.com/w/cpp/header/regex>

В строках C++ обратный слэш нужно экранировать

```
#include <regex>
using namespace std;
int main(){
    // строки, в которых будет происходить поиск по регулярному выражениям
    string str = "Hello world";
    string number = "42";
    // объект регулярное выражение: \d\d -- две цифры подряд
    regex rx("\\d\\d");

    bool result = regex_match(str, rx);    // false
    result = regex_match(number, rx);    // true
}
```

Поиск соответствия с возвратом позиции и длины

```
string str2 = "Don't Panic!";
regex rx2("Panic");

// ищет совпадение в строке
result = regex_search(str2, rx2);
cout << result << endl;

// проверяет на соответствие строку целиком
result = regex_match(str2, rx2);
cout << result << endl;
```

Поиск нескольких соответствий:

```
string str2 = "Don't Panic! Panic!";
regex rx2("Panic");
smatch m; // для сохранения информации о совпадениях

// поиск всех совпадений
while( regex_search(str2, m, rx2) ){
    // строка соответствующая выражению
    cout << m[0].str() << endl;
    cout << m.position(0) << endl;
    cout << endl;
    // берём остаток строки для продолжения поиска
    str2 = m.suffix();
}
```

```
Вывод:  
Panic  
6  
  
Panic  
2
```

## 3.8 Взаимодействие с ОС

Выполнение команды, будто она была вызвана в оболочке командной строки:

```
#include <cstdlib>  
//  
system("calc.exe");
```

Получение значений переменных окружения:

```
#include <cstdlib>  
  
char *str = getenv("PATH");
```

Некоторые переменные окружения Windows:

- COMPUTERNAME – имя компьютера;
- HOMEPATH, USERPROFILE – имя компьютера;
- TEMP, TMP – путь к папке для временных файлов;
- PATH – список путей к исполняемым файлам, доступным непосредственно (например из оболочки командной строки).

Компилятор MSVC может не компилировать программу, где вызывается функция `getenv`, так как при обращении к некорректному имени переменной окружения программа может аварийно завершиться. Для игнорирования ошибки компилятора можно использовать директиву `#pragma warning(disable:4996)`.

## 3.9 Время

Функция `time( std::time_t* arg )` из файла `<ctime>` возвращает количество секунд, прошедших с полуночи (00:00:00 UTC) 1 января 1970 года (так

стандарт языка C++ не определяет начало отсчёта и представление времени, но общепринято использование времени Unix

называемая эпоха Unix).

Если в функцию передана переменная типа `std::time_t*` то время запишется и в неё. Но допустимо вместо указателя передавать пустой указатель `nullptr`. Сам тип `std::time_t` тоже не определён стандартом, но как правило является синонимом типа `unsigned long`.

С помощью функции `time` можно организовать простейшее профилирование кода, измеряя времени выполнения отдельных частей программы.

```
#include <ctime>
//...

size_type N = 10e8;
int *arr = random_array(N);
// запоминание начала отсчёта
time_t t0 = time(nullptr);
// код, время выполнения которого нужно измерить
quicksort(arr,N);
time_t t1 = time(nullptr);
cout << "time delta (seconds) " << time(nullptr) - t0;
```

Стоит помнить, что на время выполнения программы влияет диспетчеризация процессов, которая находится в ведении ОС. Выполнение других программ может замедлить выполнение вашей программы. Поэтому для более достоверной оценки времени стоит запускать измерения несколько раз для кода, выполнение которого занимает минимум десятки секунд.

для измерения времени работы всей программы в linux можно использовать программу `time: time my_program`

Измерение времени с точностью до наносекунд возможно с помощью библиотеки `chrono`:

```
#include <chrono>
using namespace std::chrono;
// ...

size_type N = 10e8;
int *arr = random_array(N);

// начальная отметка времени
auto t0 = steady_clock::now();
quicksort(arr,N);
// конечная отметка времени
auto t1 = steady_clock::now();
// преобразование времени (обычно наносекунды) в миллисекунды
auto delta = duration_cast< milliseconds >(t1 - t0);
cout << "time delta (milliseconds) " << delta;
```



## 4 Введение в объектно-ориентированное программирование

### 4.1 Предпосылки появления ООП

todo: ...

Предметная область –

todo: Предметно-ориентированное проектирование (Domain-driven design

### 4.2 Абстрактный тип данных

Абстрактный тип данных (АТД, Abstract Data Type – ADT) – это математическая модель для типов данных, где тип данных определяется поведением (семантикой) с точки зрения пользователя данных, а именно в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций.

АТД позволяет описать тип данных независимо от языка программирования.

Но как и в языках программирования, в описании АДТ существуют договорённости по структуре и стилю описания

**ADT** НаименованиеАбстрактногоТипаДанных

- **Данные**

... перечисление данных ...

- **Операции**

- **Конструктор**

Начальные значения:

Процесс:

– **Операция...**

Вход:

Предусловия:

Процесс:

Выход:

Постусловия:

– **Операция... ..**

**Конец ADT** НаименованиеАбстрактногоТипаДанных

- Данные – набор из общих свойств для описываемой общности объектов
- Операции – действия которые можно совершать над данными
  - Вход – необходимые входные данные для совершения операции. Могут отсутствовать.
  - Предусловия – требования к входным данным, при соблюдении которых операция может быть произведена.
  - Процесс – совершаемые действия.
  - Выход – выходные данные, получаемые после совершения действия. Могут отсутствовать.
  - Постусловия – требования к данным, которые должны быть соблюдены после выполнения действия.

**Пример.** Опишем АД для простого секундомера, способного измерять время в на отрезке от 0 до 59 секунд.

**ADT Секунды**

- **Данные**  $s$  – число секунд
- **Операции**
  - **Конструктор**  
Начальные значения: 0  
Процесс:  $s = 0$

– **Операция «Задать число секунд»**

Вход:  $s1$

Предусловия:  $0 \leq s1 \leq 59$

Процесс:  $s = s1$

Выход: -

Постусловия: -

– **Операция «Прочитать число секунд»**

Вход: -

Предусловия: -

Процесс: прочитать  $s$

Выход:  $s$

Постусловия: -

– **Операция «Увеличить число секунд на единицу»**

Вход: -

Предусловия: -

Процесс:  $s = (s + 1) \% 60$

Выход: -

Постусловия: -

% – операция  
вычисления остатка от  
деления

## **Конец ADT Секунды**

**Абстрагирование.** При описании АД следует абстрагироваться от несущественных свойств описываемой сущности, не добавлять лишних данных. Какие свойства считать существенными зависит от предметной области. Например, с точки зрения геоинформационной системы (например гугл-карт) магазин должен иметь адрес, режим работы, может иметь контактные данные (номер телефона, сайт, адрес электронной почты) и фотографии. С точки зрения владельца магазина имеет более сложное устройство, включающее ассортимент, сотрудников, бухгалтерскую информацию и так далее.

## **Классы и объекты**

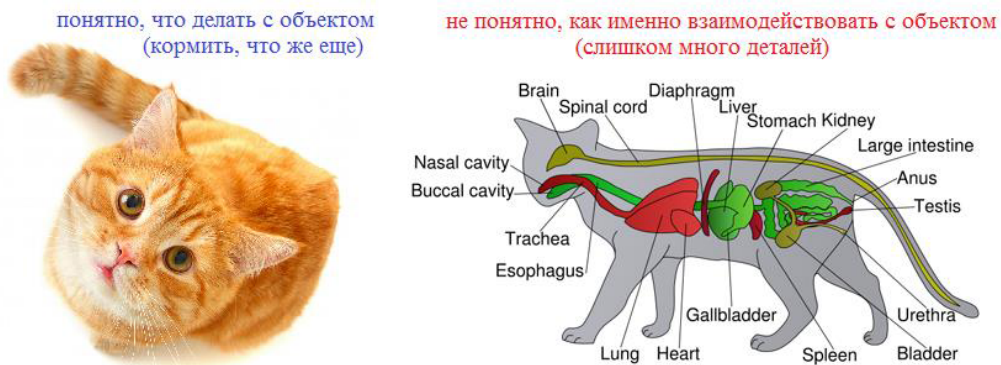


Рис. 4.1. Метафора абстрагирования с точки зрения разных предметных областей

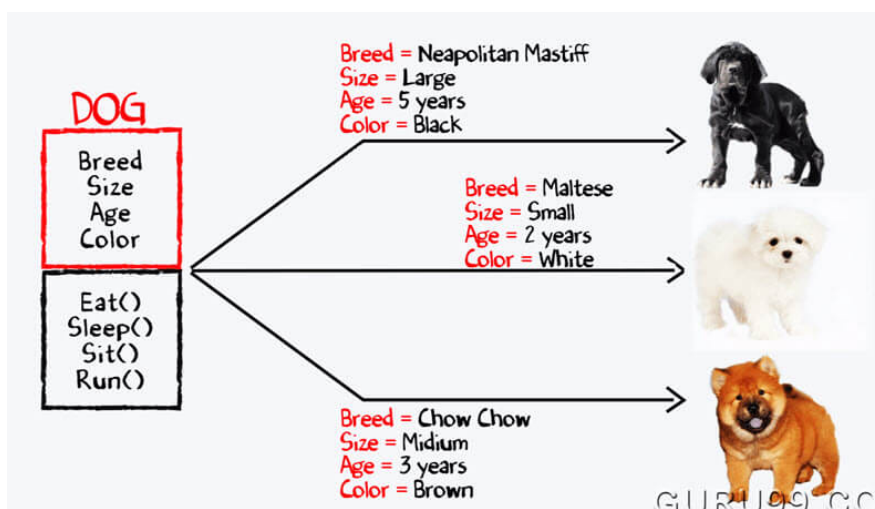


Рис. 4.2. Класс Dog и объекты

## 4.3 Классы в C++

Метод –

Поле –

Поля и методы – это члены класса.

Общий синтаксис объявления класса:

```
class_key attr(optional) ClassName
    final(optional) base_clause(optional) {
        private:
        and\or
        protected:
        or\and:
        private:

        member-specification
    };
```

объявление класса в документации языка:  
[en.cppreference.com/  
/w/cpp/language/class](http://en.cppreference.com/w/cpp/language/class)

В конце объявления класса должна стоять либо точка с запятой, либо объявлены объекты или указатели на объекты данного класса.

- `class_key` – `class`, `struct` или `union`.
- `attr` – атрибуты (не обязательно).
- `ClassName` – идентификатор (имя) класса.
- **`final`** – если присутствует, то от класса нельзя наследоваться.
- `base_clause` – имена базовых классов с модификаторами наследования.
- **`private`**, **`protected`**, **`private`** – области видимости: открытая (доступна всем), защищённая (доступна классу и потомкам), `private` (доступна классу).
- `member-specification` – описание членов класса

Возможно предварительное объявление (forward declaration) класса

```
class_key attr(optional) ClassName;
```

с последующим его определением. Это может быть полезно, если класс В использует класс А (значит А должен быть объявлен раньше), а класс А использует класс В.

Стоит избегать избыточности при создании полей класса. Если значения некоторых полей можно быстро определить на основе остальных, то стоит вместо этих полей создать методы. Например, в примере модуль комплексного числа `mod` может быть вычислен по известной действительной и мнимой частям. Кроме того, при изменении комплексного числа придётся каждый раз вычислять модуль, чтобы значения полей оставались непротиворечивыми, а отдельное изменение этого поля пришлось бы запретить.

### Пример 1. Избыточные поля

```
/// Комплексное число
class ComplexNumber{
public:
    double real, imag;  // действительная и мнимая часть числа

    double mod;        // модуль комплексного числа -- избыточное поле

    // ...
};
```

### Пример 2. Метод вместо поля

```
/// Комплексное число
class ComplexNumber{
public:
    double real, imag;  // действительная и мнимая часть числа

    // ...

    /// модуль
    double mod() const {return pow(real*real + imag*imag, 0.5) };

    // ...
};
```

В C++ разница между **struct** и **class** не существенна. В **struct** область видимости по умолчанию – `public`, в **class** – `private`.

Принцип сокрытия сложности – todo

**Инкапсуляция** (в ООП) – один из базовых принципов ООП, обозначающий объединение данных (полей) и методов работы с этими данными в одном классе. Доступ к данным, в частности их изменение, *может* происходить только через методы. Это позволяет использовать объекты не задумываясь о деталях реализации хранения данных внутри них. Кроме того, сеттеры создаются такими, чтобы поддерживать целостность данных. Например класс, отвечающий за хранение даты, не позволит задать месяц равный 13 или задать 29 февраля в не високосный год.

В C++ и других подобных языках (например Java и C#) принцип инкапсуляции тесно связан с **принципом сокрытия**. Принцип сокрытия требует разграничить доступ к полям и методам, чтобы избежать прямого доступа к некоторым или всем полям (поместить их в область private или protected) оставив возможность взаимодействовать с ними только через методы, которые объявлены в области public.

todo: ссылка на Свойства в C#

## Класс Секунд. Пример 1

Пример класса:

```
/// Класс для отсчёта секунд на отрезке от 0 до 59 с переполнением
class Seconds{
    // поле класса (по умолчанию private)
    /// количество секунд
    short s;

public: // открытый раздел класса
    // Методы:
    // конструктор по умолчанию
    Seconds() {s = 0;}

    // конструктор с параметром
    Seconds(short s1) {
        setSeconds(s1);
    }

    /// сеттер секунд
    void setSeconds(short s1){
        if ( (s1 >= 0) && (s1 <=59) )    // проверка предусловия
            s = s1;
    }

    /// геттер секунд
    short seconds() const {return s;}
}
```

```

    /// возвращает строковое представление данных класса
    std::string toString() const {return std::to_string(s);}

    /// увеличение секунд на 1.
    /// при переполнении возвращает результат по модулю 60
    short tick() {
        s = (s+1) % 60;
        return s;
    }
};

```

Поле класса **short** **s**; объявлено в закрытой области видимости (**private**) для его защиты от непосредственного доступа и записи некорректных значений (принцип сокрытия). Для чтения и изменения (с проверкой предусловий) этого поля создаются отдельные методы – геттер и сеттер соответственно.

Конструктор `Seconds()` `{s = 0}` – специальный метод класса, который инициализирует его поля. Этот метод вызывается после создания экземпляра класса. Для него не указывается возвращаемый тип данных (в отличие от остальных методов), так как этот метод всегда возвращает тип соответствующий своему классу (`Seconds`).

Конструктор с параметром `Seconds(short s1)` полезен, если нужно сразу задать значения полей класса. В таких конструкторах стоит вызывать другие методы – сеттеры, так как они уже содержат проверки предусловий.

**void** `setSeconds(short s1)` – сеттер для секунд. Принимает новое значение для секунд, проверяет его (проверка предусловия) и задёт, если новое значение не противоречит логике класса.

**short** `seconds()` **const** – геттер для секунд. **const** означает, что метод не изменяет поля класса. Такой метод можно вызывать для объекта константы.

`std::string toString()` **const**. Для вывода объектов на экран в частности и для преобразования объекта в строку стоит создавать в классе отдельный метод, возвращающий в том или ином виде текстовое представление состояния класса.

Так как класс является типом данных, можно создать переменные этого типа – **объекты (экземпляры класса)**:

```

// создание экземпляра класса, вызов конструктора по умолчанию
Seconds s1;

```



```

Seconds s01();

// создание экземпляра класса, вызов конструктора с параметром
Seconds s2(42);

// Динамическое создание экземпляра класса
Seconds *s3 = new Second();
Seconds *s4 = new Second(42);

s3.setSeconds(20);

cout << s1.seconds() << endl;    // 0
cout << s2.seconds() << endl;    // 42
cout << s3->seconds() << endl;    // 20
cout << s4->seconds() << endl;    // 42

```

Классы принято объявлять в заголовочных файлах, а реализацию методов в cpp файле. Имена этих файлов как правило совпадают с именем класса. При определении метода в отдельном файле его полное имя состоит из имени класса и имени метода внутри класса:

```

// определение метода
return_type ClassName::method_name(args) attributes;

```

Изменим пример из параграфа 4.3, разделив объявления на два файла.

## Класс Секунд. Пример 2.

```

// определение ранее объявленного метода
return_type ClassName::method_name(args) attributes;

```

**seconds.h** – объявление класса:

```

#pragma once
#include <string>

/// Класс для отсчёта секунд на отрезке от 0 до 59 с переполнением
class Seconds{
    // поле класса (по умолчанию private)
    /// количество секунд
    short s;

public: // открытый раздел класса
    // Методы:
    // конструктор по умолчанию
    Seconds();

    // конструктор с параметром
    Seconds(short s1);

```

```

    /// сеттер секунд
    void setSeconds(short s1);

    /// геттер секунд
    short seconds() const;

    /// возвращает строковое представление данных класса
    std::string toString() const;

    /// увеличение секунд на 1.
    /// при переполнении возвращает результат по модулю 60
    short tick();
};

```

**seconds.cpp** – определения методов класса:

```

#include "seconds.h"          // файл с объявлением класса Seconds

// конструктор по умолчанию
Seconds::Seconds() {
    s = 0;}

// конструктор с параметром
Seconds::Seconds(short s1) {
    setSeconds(s1);
}

/// сеттер секунд
void Seconds::setSeconds(short s1){
    if ( (s1 >= 0) && (s1 <=59) )    // проверка предусловия
        s = s1;}

/// геттер секунд
short Seconds::seconds() const {
    return s;}

    /// возвращает строковое представление данных класса
std::string Seconds::toString() const {
    return std::to_string(s);}

    /// увеличение секунд на 1.
    /// при переполнении возвращает результат по модулю 60
short Seconds::tick() {
    s = (s+1) % 60;
    return s;}

```

**Локальный класс** – класс, определённый внутри функции.

## Объекты-константы

Объекты, которые занимают память большую чем размер указателя, эффективно передавать в функции по ссылке. Если изменение формального параметра внутри функции не требуется, то для защиты от непреднамеренного изменения такие объекты стоит делать константами внутри функции.

Например

```
Seconds seconds_plus(const Seconds& a, const Seconds& b){
    Seconds c;
    c.setSeconds( ( a.seconds() + b.seconds() ) % 60 );
    return c; }
```

Если метод seconds() объявлен как без спецификатора const:

```
short Seconds::seconds() { ... }
```

то его вызов для объекта-константы запрещён, ибо компилятор не может гарантировать, что этот метод не изменяет объект.

Все методы со спецификатором const можно вызывать для объектов констант

```
short Seconds::seconds() const { ... }
```

### 4.3.1 Конструкторы и деструктор

**Конструктор** – это особый метод, инициализирующий экземпляр своего класса.

- Имя конструктора совпадает с именем класса<sup>1</sup>.
- Тип возвращаемого значения не указывается.
- У конструктора может быть любое число параметров.
- У класса может быть любое число конструкторов.
- Конструкторы могут быть доступными (public), защищенными (protected) или закрытыми (private).

---

<sup>1</sup>конструктор в python называется \_\_init\_\_

- Если не определено ни одного конструктора, компилятор создаст конструктор по умолчанию, не имеющий параметров (а также некоторые другие к. и оператор присваивания)

## Виды конструкторов

- Конструктор умолчания (default constructor): `ClassName()`
- Конструктор с параметрами:
  - конструктор преобразования (conversion constructor):  
`ClassName( arg )`
  - конструктор с двумя и более параметрами (parameterized constructors):  
`ClassName( arg1, arg2, ... )`
- конструктор копирования (copy constructor):  
`ClassName( const ClassName& )`
- конструктор перемещения (move constructor):  
`ClassName( const ClassName && )`

## Конструктор с параметрами (parameterized constructor)

Конструктор с несколькими параметрами упрощает создание и инициализацию объекта. При этом такой конструктор должен не допускать задания некорректных значений полей, поэтому в нём нужно вызывать сеттеры, а не напрямую задавать значения:

```
/// Книга
class Book{
    string title;           // заглавие
    unsigned pages;        // количество страниц
    // ...

    Book(){ title = ""; pages = 0;}

    // конструктор с параметрами
    Book(string title1, unsigned pages1) {
        set_title(title1);
        set_pages(pages1);
    }

    // ...
};

Book b1;                  // вызов конструктора по умолчанию
```

```
// вызов конструктора с параметрами:  
Book b2("Незнайка на Луне", 600);  
Book b3 = Book("1984", 300);
```

## Конструктор преобразования (Conversion constructor)

.

Общий вид:

```
ClassName(T t)
```

T – некоторый тип

- Принимает один параметр
- Тип параметра должен отличаться от самого класса
- Такой конструктор как бы преобразует один тип данных в экземпляр данного класса
- Может вызываться при инициализации объекта значением принимаемого типа

```
ClassName c = t
```

Примеры вызова конструктора (для класса описанного в разделе 4.3):

```
// способы вызова конструктора преобразования  
Seconds s1(42);  
Seconds s2 = Seconds(42);  
Seconds s3 = 42;  
Seconds s4 {42};  
  
// Ошибка: нет конструктора с параметром типа double  
Seconds s5 = 42.0;
```

На месте типа, который передаётся в конструктор может быть в том числе другой класс.

## Правило пяти

Если класс или структура определяет один из следующих методов, то нужно явным образом определить все методы:

- Конструктор копирования

- Конструктор перемещения
- Оператор присваивания копированием
- Оператор присваивания перемещением
- Деструктор

**Поверхностное копирование** (shallow copy) – копирует только объект, а состояние является разделяемым. Автоматически сгенерированный конструктор копирования способен выполнить только поверхностное копирование.

**Глубокое копирование** (deep copy) – копирует объект и состояние, если нужно — рекурсивно.

**Пример.** Приведём частичную реализацию класса, реализующего динамический массив (аналог vector).

```

/// Динамический массив
class Array{
    float *arr;
    unsigned n;      /// размер массива

public:
    // конструктор по умолчанию
    Array(){ n = 0; arr = nullptr;}

    /// констр. создающий массив из n1 нулевых элементов
    Array(unsigned n1){
        // проверка предусловия
        if (n1 < 0) throw std::length_error("Error: size <= 0");
        n = n1;
        arr = new float[n] {0}; }

    // ...

    ~Array(){
        if (arr != nullptr)
            delete []arr;
    }
};

```

Такой класс упростит работу с динамическими массивами. Не придётся заботиться об освобождении памяти массива. Перед завершением функции process она очистит память, занимаемую всеми локальными переменными. Перед удалением локального объекта а будет вызван её деструктор для деинициализации. Деструктор освободит память, занимаемую массивом.

```

void process(){
    Array a(10); // вызов констр, выделение памяти под 10 элементов

    // работа с массивом
    // ...

    // неявный вызов деструктора: a.~Array()
}

```

Согласно правилу пяти, нужно реализовать несколько дополнительных методов, которые обычно создаются компилятором. **Конструктор копирования.** При создании копии экземпляра Array нужно чтобы новый экземпляр имел свою область памяти отведённую под массив во избежание конфликтов. Если оба объекта будут ссылаться на один и тот же массив, то, например при удалении, первого объекта, вызовется его деструктор. Который освободит память, на которую ссылается указатель внутри второго объекта. Значит нужно реализовать конструктор копирования, который будет выделять память под свой массив и копировать туда значения из существующего.

```

Array a(10);
Array b = a;

```

По тем же причинам нужно реализовать **оператор присваивания копированием.** В дополнении, объект, которому присваивают новое значение должен освободить память, занимаемую своим массивом чтобы избежать её утечки.

```

Array a(10);
Array b(20);
b = a; // утечка памяти из b;

```

...

## Порядок вызова конструкторов и деструкторов

В момент выполнения собственного конструктора все информационные поля должны быть уже проинициализированы.

После создания объекта, порядок вызова конструкторов:

1. **Конструкторы базовых классов** в порядке их появления в описании класса. Если в списке инициализации описываемого класса присутствует вызов конструктора преобразования или конструктора с двумя

и более параметрами) базового класса, то вызывается конструктор преобразования (link) или конструктор с двумя и более параметрами), иначе вызывается конструктор умолчания базового класса.

2. **Конструкторы** умолчания всех **полей**, которые не перечислены в списке инициализации, и конструкторы преобразования, копирования и конструкторы с двумя и более параметрами всех полей, из списка инициализации. Все перечисленные в данном пункте конструкторы умолчания, преобразования, копирования, с двумя и более параметрами) вызываются в порядке описания соответствующих информационных членов в классе.
3. Собственный конструктор.

Условия вызова деструктора:

1. свёртка стека – при выходе из блока описания объекта, в частности, при обработке исключений при выходе из try-блока по оператору throw, try-блоки описываются далее), завершении работы функций;
2. при уничтожении временных объектов – сразу, как только завершится конструкция, в которой они использовались;
3. при выполнении операции delete для указателя, получившего значение в результате выполнения операции new. После выполнения деструктора освобождается выделенный для объекта участок памяти;
4. при завершении работы программы при уничтожении глобальных и статических объектов.

Порядок вызова деструкторов:

1. Собственный деструктор. В момент начала его работы поля класса еще не очищены, и их значения могут быть использованы в теле деструктора.
2. Деструкторы вложенных объектов в порядке, обратном порядку их описания.
3. Деструкторы базовых классов в обратном порядке их задания.

**Инициализация членов класса**

```
class Example1{  
    float x = 0.0;
```



```

int *y = nullptr;
const int z = 42;

public:
    Example1() { }
};

```

**Списки инициализации полей класса.** Присваивание значений константным или ссылочным переменным-членам в теле конструктора в некоторых случаях невозможно. Проблему решает специальный вид инициализации, который происходит до вызова тела конструктора:

```

class Example2{
    float x;
    int *y;
    const int z;

public:
    Example2()
        // список инициализации:
        : x(0.0), y(nullptr), z(42)
    {
        // ...
    }
};

```

## спецификаторы default и delete

### 4.3.2 Работа с экземплярами класса

```

// создание объектов
Seconds s1, s2(12);
// объект s1 создаётся с вызовом конструктора по умолчанию
// объект s2 создаётся с вызовом конструктора преобразования (с одним параметром)

Seconds s1234 = 12; // к. преобразования

// вызов операторов
Seconds s3 = s1 + s2++;
cout << "секунды: " << s3.toString() << endl;

Seconds *s = new Seconds; // динамическое создание
s->setSeconds(10);

// ссылка на объект
Seconds &s10 = s1;
delete s;

```

Объекты одного класса будут считаться **равными**, если они имеют одинаковые значения своих полей. В примере выше объекты s2 и s1234 равны.

Также равны объект s1 и объект, на который указывает s, так как они были инициализированы с помощью вызова конструктора по умолчанию, который всегда задаёт одно и то же значение единственному полю класса.

Объекты s10 и s1 **идентичны**, так как они представляют собой одну и ту же область памяти.

Массивы из объектов:

```
// Массив из 128 объектов. Для каждого вызван к. по умолчанию
Seconds ss1[128];

// Массив из 3 объектов. Вызваны конструкторы: по молчанию, преобразования, по умолчанию
Seconds ss2[] = {Seconds(), Seconds(2), Seconds};

// Обращение к членам класса через массив
ss1[42].setSeconds(15);

// Статический массив из указателей на объекты
Seconds * ss3[2];
// Выделение памяти под каждый объект
ss3[0] = new Seconds();
ss3[1] = new Seconds();
// Обращение к членам класса:
ss3[0]->setSeconds(15);
int n = ss3[0]->getSeconds();
// Освобождение памяти
delete ss3[0];
delete ss3[1];
```

Динамические массивы из объектов:

```
// Выделение памяти под 512 объектов, для каждого вызывается конструктор по умолчанию
Seconds *ss4 = new Seconds[512];

ss4[5].setSeconds(16);

vector<Seconds> vec1; // динамический массив из объектов
// добавлении объектов в массив
vec1.push_back(s1);
vec1.push_back(s2);

for (int i=0; i<10; i++){
    Seconds s( rand() % 60 );
    vec1.push_back(s);
}

cout << "Секунды:" << endl;
for (Seconds s : vec1){
    cout << s.toString() << endl;
}
```

```
vec1[1].setSeconds(20); // обращение к отдельному элементу вектора. вызов метода
Seconds s5 = vec1[1];
```

```
vector<Seconds*> vec2; // набор из указателей на объекты
// создание одного объекта и вызов конструктора с параметром
Seconds *s4 = new Seconds(42);
vec2.push_back(s4);
vec2.push_back(new Seconds(43));
```

```
for (int i=0; i<10; i++){
    Seconds *s = new Seconds( rand() % 60 );
    vec2.push_back(s);
}
```

```
cout << "Секунды:"<< endl;
for (Seconds *s : vec2){
    cout << s->toString() << endl;
}
```

```
vec2[1]->setSeconds(20); // обращение к отдельному элементу вектора. вызов метода
Seconds *s6 = vec2[1];
```

```
// удаление динамически созданных объектов
for (int i=0; i<vec2.size(); i++){
    delete vec2[i];
}
```

## Временные безымянные объекты

```
// Создание объекта без имени, вызов метода, вывод на экран, уничтожение объекта
cout << Seconds(17).to_string();
```

```
// динамическое создание объекта без имени, вызов конструктора с параметром,
// вызов метода, утечка памяти!
```

```
cout << ( new Seconds(17) )->to_string();
```

*// указатель на созданный объект нигде не сохранён, поэтому нельзя освободить занимаемую им память*

## Временные безымянные объекты, возвращаемые из функции

```
Seconds get_some_seconds(){
    return Seconds();
}
```

```
// вызов метода to_string для временного безымянного объекта типа Seconds
string s = get_some_seconds().to_string();
```

```
Seconds s1(18);
```

```
// вызов метода c_str для временного безымянного объекта типа string
char * str = s.to_string().c_str();
```

## Безымянные объекты в работе со стандартными контейнерами

```
vector<Seconds> arr;  
// Создание объекта типа Seconds, вызов конструктора по умолчанию  
arr.push_back( Seconds() );  
// Создание объекта типа Seconds, вызов конструктора с параметром  
arr.push_back( Seconds(19) );  
  
// динамический массив из указателей на объекты  
vector<Seconds *> arr2;  
// Создание объекта типа Seconds, вызов конструктора по умолчанию  
arr2.push_back( new Seconds() );  
// Создание объекта типа Seconds, вызов конструктора с параметром  
arr2.push_back( new Seconds(19) );  
// освобождение памяти  
for (unsigned i = 0; i < arr2.size(); i++)  
    delete arr[i]
```

**Простая структура данных** (англ. plain old data, POD) – тип данных, имеющий жёстко определённое расположение полей в памяти, не требующий ограничения доступа и автоматического управления. Переменные такого типа можно копировать простыми процедурами копирования участков памяти наподобие memcpy. Противоположность – управляемая структура данных.

В C++ класс должны удовлетворять нескольким условиям (все поля открыты, нет виртуальных методов и д.р. [cppreference]), чтобы считаться простой структурой данных. В большинстве, созданные классы не подходят под эти требования. Поэтому обращение к области памяти, в которой расположен объект, как к набору байт, не тождественно аналогичному обращению к данным класса:

```
class NotPOD{  
public:  
    short z;           // 2 байта  
    float y;           // 4 байта  
    char get_x() { return x; }  
private:  
    char x = '#';      // 1 байт  
};  
  
NotPOD o1;  
cout << o1.get_x() << endl;      // #  
// преобразование объекта в массив байт:  
char* c = (char*)&o1;  
// обращение к полю x, которое следует за полем z и y:  
cout << c[2+4] << endl;          // Undefined Behavior!
```

При этом, для ОС разной разрядности (например 32 или 64 бита) и в зависимости от параметров компиляции объекты могут занимать больше памяти, чем чем их поля в сумме:

```
sizeof(o1);    // 12
sizeof(float)  // 4
sizeof(short)  // 2
sizeof(char)   // 1
```

Это может происходить из-за выравнивания полей в памяти. В данном примере каждому полю выделено 4 байта памяти.

Память, которую занимает объект определяется

- полями класса + остаток для выравнивания (по умолчанию выравнивание 4 байта)
- указателем на vtable (если есть виртуальные функции)
- указателями на базовые классы, от которых было сделано виртуальное наследование (размер указателя \* количество классов)

#### 4.3.3 Статические члены класса (static members)

Статические члены класса доступны без объявления объекта.

Ключевое слово **static** приводится только при объявлении члена класса:

```
static data-member;
static data-member;
```

про другие способы  
использования  
ключевого слова static  
см. раздел 2.2.1  
Статические локальные  
переменные

документация:  
[en.cppreference.com/w/cpp/la](http://en.cppreference.com/w/cpp/la)

Пример:

```
class X
{
    public:    // объявление статических членов
        static void foo();
        static int N;

        // Ошибка: определять значения в классе нельзя
        static double e = 2.718281828459045;

        // простое поле
        double tau= 1.618033988749895;

        // статическое поле-константа
        const static double Tau;
```

```

    //
    constexpr static double A = 42;

};

// определение статического поля
int X::N = 0;

// определение статического поля-константы (static в определении не указан)
const double X::Tau = 1.618033988749895;

// определение статического метода
void X::foo()
{
    // обращение к статическому полю из статического метода
    N = rand();
    // образение X::n здесь запрещено
}

void bar()
{
    // вызов статического метода
    X::foo();
    X.foo(); // ошибка!

    // создание объекта, вызов статического метода, уничтожение объекта
    X().foo();

    X x;
    x.foo(); // ошибка!

    X::N = 42;
    X().N = 43;

    X::A = 0; // ошибка!
}

```

Static member functions cannot be virtual, const, volatile, or ref-qualified.

The address of a static member function may be stored in a regular pointer to function, but not in a pointer to member function.

There is only one instance of the static data member in the entire program with static storage duration

Класс со статическим полем например может отслеживать количество своих экземпляров.

Метод или поле имеет смысл сделать статическим, если они не должны зависеть от обычных методов и полей.

#### 4.3.4 Дружественные функции и классы

Дружественные данному классу функция или другие имеют полный доступ к его закрытым членам.

```
class A{
    int data; // private data member

    // предварительное объявление класса X,
    // который имеет доступ к закрытым членам класса A
    friend class X;

    // Y -- дружественный для X класс
    friend Y;
};
```

Дружественные функции

```
class A{
    float x;

    // объявление дружественной функции
    friend void foo(A& a);
};

// определение дружественной функции
void foo(A& a){
    // функция-друг имеет доступ к закрытым членам класса
    a.x = 22./7;
};
```

#### 4.3.5 Перегрузка операторов

О перегрузке оператором см. также раздел 2.2.4 Перегрузка операторов (operator overloading).

Перегрузка оператора сложения для класса Seconds<sup>2</sup>. Оператор сложения может быть определён как<sup>3</sup>:

```
T operator+(const T &a, const T2 &b);
```

Он принимает два аргумента (операнда), типы которых могут отличаться и возвращает новое значение, тип которого совпадает с первым операндом.

**seconds.h**

---

<sup>2</sup>полный пример: [github.com/VetrovSV/OOP/tree/master/examples/simple\\_class](https://github.com/VetrovSV/OOP/tree/master/examples/simple_class)

<sup>3</sup>[en.cppreference.com/w/cpp/language/operator\\_arithmetic](https://en.cppreference.com/w/cpp/language/operator_arithmetic)

```

/// Класс для отсчёта секунд на отрезке от 0 до 59 с переполнением
class Seconds{
    /// количество секунд
    short s;

    // ...

    friend Seconds operator + (const Seconds& s1, const Seconds &s2);
};

```

## seconds.cpp

```

#include "seconds.h"

// ...

friend Seconds operator + (const Seconds& s1, const Seconds &s2);

```

Пример вызова оператора:

```

Seconds sa(55), sb(13), sc;

sc = sa + sb;
cout << sc.toString();      // 8

```

## Оператор сложения как член класса **seconds.h**

```

class Seconds{
    /// количество секунд
    short s;

    // ...

    Seconds Seconds::operator +(const Seconds& s2) const;
}

...

```

### 4.3.6 Представление класса в UML

Полный исходный код:

[github.com/VetrovSV/OOP/tree/master/examples/simple\\_class](https://github.com/VetrovSV/OOP/tree/master/examples/simple_class).

Подробно о диаграммах классов рассказывается в главе 4 Классы книги «Язык UML Руководство пользователя» [3]



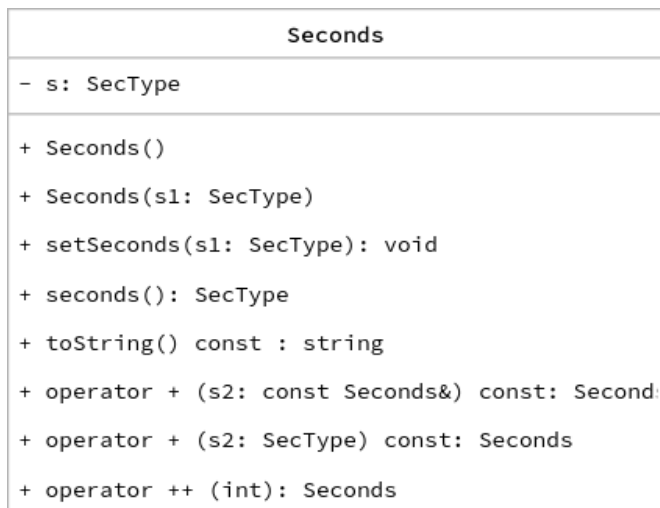


Рис. 4.3. Диаграмма класса в нотации UML. Обозначения:

- + открытый член класса (public),
- закрытый член класса (private),
- # защищённый член класса (protected)

## Порядок вызова конструкторов и деструкторов

### 4.4 Шаблонные классы

```
template < parameter_list >
class-declaration
```

Пример:

```
/// Шаблонный класс-обёртка для динамического массива
template <typename Element>
class Array{
    Element *arr = nullptr;
    unsigned n = 0;
public:

    Array(){};

    Array(unsigned n1){
        if (n1 > 0) {
            n = n1;
            arr = new Element[n]; }
        // ...
    }

    // геттер для элемента массива
    Element get(unsigned i) const {
        if (i < n) return arr[i];
    }
    // сеттер для элемента массива
    void set(unsigned i, Element el){
```

```

        if (i < n) arr[i] = el;
    }

    // ...
};

int main(){

    // компилятор создает реализация класса с Element = int
    Array<int> arr1 (128);
    arr1.set(0, 1729);

    // компилятор создает реализация класса с Element = int
    Array<double> arr2;

    // Ошибка: имя шаблонного типа не указано!
    Array arr0;
}

```

См. также раздел 2.3 Шаблонные функции.

## 4.5 Модульное тестирование

**Модульный автоматический тест** (unit test) – Пример тестирования класса [github.com/VetrovSV/OOP/tree/master/examples/simple\\_class](https://github.com/VetrovSV/OOP/tree/master/examples/simple_class)

Руководство Google Test: <http://google.github.io/googletest/primer.html>

Фреймворк Google Test содержит набор макросов для всевозможных проверок (замены макросу `assert` из стандартной библиотеки C++). Отдельные тестовые случаи предлагается описывать внутри специальных тестовых функций (см. пример ниже). Эти тестовые функции легко интегрируются в средства выполнения тестов сред разработки и могут быть запущены независимо от основного проекта. Сами средства разработки зачастую содержат шаблоны проектов для тестирования.

### Интеграция модульных тестов Google Test в Visual Studio

1. Открыть решение, для которого нужно сделать модульное тестирование
2. Добавить *проект* теста в решение:
  - (а) Файл > Добавить > Создать проект > Выбрать *шаблон Google Test* (фильтр проектов – Тестирование)
  - (б) Задать название проекта тестов, выбрать проект, для которого будут написаны тесты

3. В созданном проекте будет один файл исходных текстов с примером тестового макроса (функции):

```
TEST(TestCaseName, TestName) {  
    // примеры макросов:  
    EXPECT_EQ(1, 1);        // макрос проверки равенства  
    EXPECT_TRUE(true);     // макрос проверки истинности  
}
```

TestCaseName – идентификатор тестируемого класса или модуля

TestName – идентификатор, объясняющий назначение теста

4. Подключите зависимые файлы из основного проекта в тестовый проект (добавить > существующие ...), подключите заголовочные файлы в файл с тестами
5. Создайте тестовые макросы для вашего класса. В каждом тестовом макросе сделайте несколько проверок одного, или нескольких, логически связанных методов класса. Пример:

```
// проверка конструкторов  
TEST(Seconds, Constructor) {  
    Seconds s1;  
    EXPECT_EQ(s1.get_s(), 0);  
  
    Seconds s2(17);  
    EXPECT_EQ(s2.get_s(), 17);  
    // ...  
}  
// проверка сеттера и геттера секунд  
TEST(Seconds, AddSeconds) {  
    Seconds s;  
    s.add_seconds(19)  
    EXPECT_EQ(s.get_s(), 19);  
  
    // todo: -1, 0, 59, 60  
}
```

Создавайте отдельные экземпляры класса для разных тестов, чтобы упростить тестирование и избежать перекрёстного влияния методов друг на друга

6. Откройте список тестов и запустите их: меню Тест > Обозреватель тестов

Проверка брошенного исключения

```
Seconds s;  
try {  
    s.setSeconds(100);  
    FAIL() << "Expected InvalidValue";  
}  
catch(SecondsException & err) {  
    EXPECT_EQ(err, InvalidValue);  
}  
...
```

## Некоторые макросы

Макросы вида `ASSERT_*` используются когда неудавшаяся проверка должна остановить выполнение тестового проекта, макрос `EXPECT_` с неудавшейся проверкой не отменяет выполнение следующих за ним проверок. Эти два варианта макросов имеют парные варианты проверок, поэтому приведём только `EXPECT`-макросы:

- Проверка логических выражений `EXPECT_TRUE( condition )`, `EXPECT_FALSE( condition )`
- Проверка равенств и неравенств: `EXPECT_EQ( x, y )`, `EXPECT_NE( x, y )` и др. с суффиксами: `EQ` – равно, `NE` – не равно, `LT` – меньше, `LE` – меньше или равно, `GT` – больше, `GE` – больше или равно.
- Проверка C-строк `EXPECT_STREQ(str1, str2)`, `EXPECT_STRNE`
- Сравнение вещественных чисел `EXPECT_FLOAT_EQ(val1, val2)`, `EXPECT_DOUBLE_EQ`
- Сравнение вещественных с заданной точностью `EXPECT_NEAR(val1, val2, abs_error)`
- Проверка исключений: `EXPECT_THROW(statement, exception_type)` – ожидание исключения типа `exception_type`

Подробнее: <http://google.github.io/googletest/reference/assertions.html>

### Дополнительно:

- [youtube.com/watch?v=6pp8S56sS2Y](https://www.youtube.com/watch?v=6pp8S56sS2Y) – Настройка и использование Google Test в Qt Creator
- <https://doc.qt.io/qtcreator/creator-autotest.html> – тестирование в Qt Creator
- CLion: Unit testing tutorial

См. также

- Статические анализаторы кода: `cppcheck`, `PVS-studio`, встроенные в Qt Creator, Visual Studio, CLion
- Контрактное программирование (появилось в C++20)
- Метаморфное тестирование – <https://habr.com/ru/post/454458/>

# 5 Отношения между классами

## 5.1 Ассоциация

## 5.2 Агрегация и композиция

**Композиция** – отношение при котором один класс (часть, Part) является неотъемлемой частью другого класса (целое, whole). В классе Whole может быть поле типа Part:

```
class Part{
    // ...
};

class Whole{
    Part filed1;
    // ...
};
```

В классе Whole может быть поле типа указатель на Part, но при этом класс Whole также владеет объектами Part создавая их в конструкторе и уничтожая в деструкторе:

```
class Part{
    // ...
};

class Whole{
    Part *filed1;
    // ...

    public:
        Whole(){ filed1 = new Part();}

        ~Whole(){ delete filed1;}
};
```

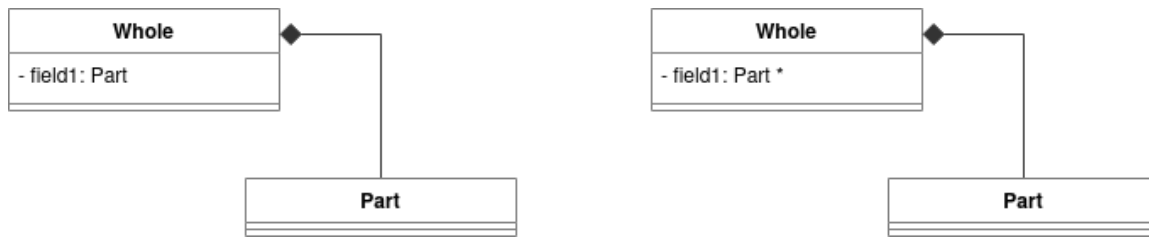


Рис. 5.1. Отношение Композиция

Агрегация – более слабый вариант композиции, когда время жизни части и целого не связаны. Можно создать объект Whole не создавая объекта Part.

```

class Part{
    // ...
};

class Whole{
    Part *field1;
}
  
```

Класс Whole может хранить в себе указатель на Part, но экземпляр класса Whole можно создать и отдельно; Время жизни Part не привязано к Whole.

### Мощность отношения – todo

Композиция:

```

class Part{
    // ...
};

class Whole1{
    Part field1[10];
}
  
```

Композиция:

```

class Part{
    // ...
};

class Whole2{
    Part *field1;           // указатель на массив
    // или
    vector< Part > field2;
    // или
    vector< Part* > field3; // массив из указателей
    // ...
}
  
```

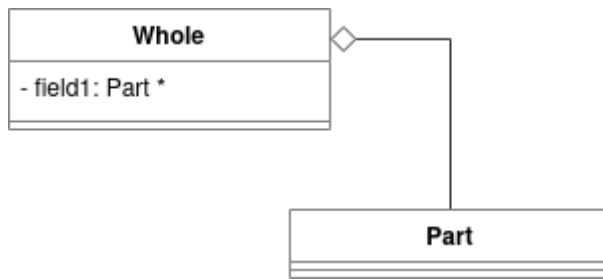


Рис. 5.2. Отношение Агрегация

**Ассоциация** – самый общий вид отношений между классами, когда один класс неким образом связан с другим, при этом не является частью (агрегация или композиция) или частным случаем (наследование) другого.

Примеры ассоциации через использование одним классом другого класса в методе

```

class Foo{
    // ...
};

class Bar{
    public:
        void baz( Foo f ) {...}
}
  
```

```

class Foo{
    // ...
};

class Bar{
    public:
        void baz( ) {
            Foo f;
            ...
        }
}
  
```

Класс Programmer содержит внутри себя указатель на Manager, однако последний по смыслу не является частью первого, но программист знает о своём менеджере. И наоборот Manager по смыслу не состоит из Программистов, но знает про программистов в своём подчинении. Такая взаимная ассоциация называется двунаправленной. На диаграмме классов изображается линией.

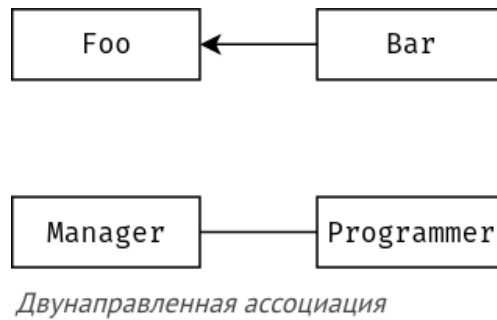


Рис. 5.3. Однонаправленная и двунаправленная ассоциация

```
// предварительное объявление класса Manager для использования в другом классе
class Manager;

class Programmer{
    Manager *manager;
    // ...
};

class Manager{
    vector<Programmer*> programmers;    // массив из указателей на программистов
    // ...
}
```

## 5.3 Наследование (inheritance)

**Пример. Один класс.** Опишем класс для вектора  $V = (v_x, v_y)$  на плоскости

```
/// Вектор на плоскости
class Vector2D{
protected:
    float _x, _y;
public:
    Vector2D() { _x = 0; _y = 0; }
    Vector2D(float x, float y) { _x = x; _y = y; }

    void setX(float x) { _x = x; }
    void setY(float y) { _y = y; }

    float x() const { return _x; }
    float y() const { return _y; }

    /// длина вектора
    float abs() const { return sqrt(_x*_x + _y*_y); }
};
```

члены класса с модификатором доступа protected доступны только классу и его потомкам (производным классам)



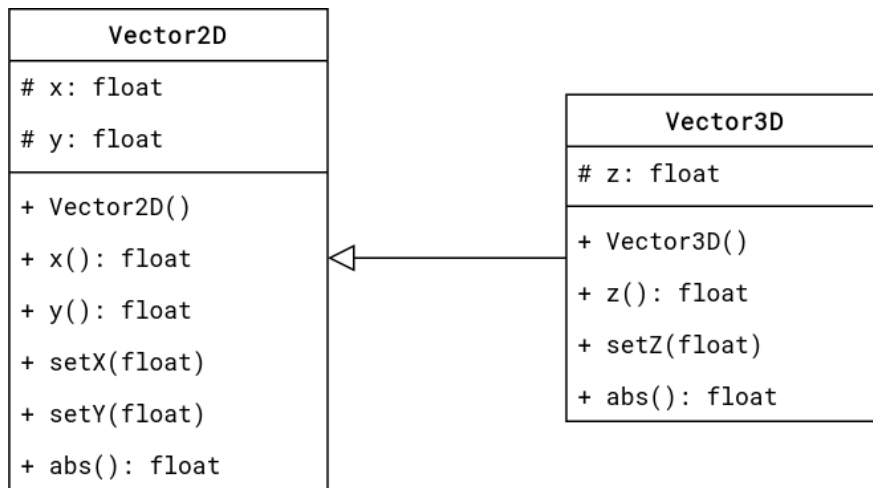


Рис. 5.4. Наследование на UML диаграмме классов. Для Vector3D (производный класс) приведены только его собственные поля, без полей унаследованных от Vector2D (базовый класс)

Требуется создать ещё класс для представления вектора в пространстве  $V = (v_x, v_y, v_z)$ . Новый класс можно построить на основе старого, в котором уже будут методы существующего класса.

**Наследование (inheritance)** – построение новых классов на основе существующих.

**Базовый класс (предок)** – класс на основе которого строится определение нового класса – **производного класса (потомка)**, см. рис 5.4. Производный класс ещё называют **подклассом** (subclass) и иногда *подтипом* (subtype), однако на счёт точного значения последнего термина консенсуса нет.

При наследовании в новый класс переходят все поля базового класса и все его не специальные методы вне зависимости от модификатора доступа.

специальные методы: конструкторы, деструктор, операторы присваивания

Создадим производный класс Vector3D на основе базового Vector2D:

```
/// Вектор в пространстве
class Vector3D : public Vector2D{

protected:
    /// поля _x и _y унаследованы
    float _z;
public:
    /// x(), setX() и др. методы тоже унаследованы

    Vector3D() { /// можно вызывать к. базового класса:
        Vector2D();
        _z = 0; }

    Vector3D(float x, float y, float z)
        : Vector2D(x,y)    /// другой способ вызова к. базового класса
        { _z = z; }        /// тело конструктора данного класса

    void setZ(float z) { _z = z; }
    float z() const { return _z; }

    /// метод вычисления длины вектора здесь должен быть переопределён
    float abs() const {
        return sqrt(_x*_x + _y*_y + _z*_z); }
};
```

**public** – модификатор наследования (подробнее см. ниже). Базовый класс должен быть объявлен раньше чем производный, например в подключаемом заголовочном файле.

У класса Vector3D метод вычисления длины abs должен иметь свою реализацию. Поэтому в классе метод **переопределяется** (ovverride), затеняя собой одноимённый метод базового класса.

Создание объекта и вызов методов:

```
Vector3D v1;

v1.setX(3);    /// вызов унаследованного метода
v1.setZ(4);

/// вызов переопределённого метода:
v1.abs();      /// 5
```

Для классов связанных наследованием возможно автоматическое (неявное) преобразование типов:

```
Vector2D v1(10, 20);
Vector2D *v11;
```

```

Vector3D v2(100, 200, 300);
Vector3D *v22;

// так можно. но все, что не входит в Vector2D будет отброшено,
// т.к. правая часть будет преобразовано к типу Vector2D:
v1 = v2;

v11 = &v2; // и так можно.
v11->setX(42); // v2 = (42, 200, 300). но z так не поменять

// а так нельзя: откуда взять z?
v2 = v1;
// это тоже нельзя
v22 = v11;

```

При записи экземпляра производно базового класса в экземпляр базового происходит неявное преобразование типов, лишние поля отсекаются.

Запись экземпляра базового класса в экземпляр производного невозможна.

todo: память объектов производного класса.

**Наследование специальных методов.** Эти методы хоть и наследуются, но не избавляют от написания аналогичных в производном классе:

- Конструкторы
- Деструктор
- Операторы присваивания

Например в конструкторе производного класса можно вызывать конструктор базового класса, но нельзя вызывать второй *вместо* первого.

**Порядок вызова конструкторов и деструкторов.** Конструктор по умолчанию базового класса вызывается автоматически перед вызовом конструктора производного класса. Если базовых классов несколько (многоуровневое наследование) то сначала вызывается конструктор самого базового класса. Деструкторы вызываются в обратном порядке: от производного класса к базовым.

**Наследование и операторы.** Если в базовом классе был определён оператор, например сложения. То вызвать его для объекта производного класса нельзя:

```

class Vector2D{
public:

```

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

h!

Рис. 5.5. Модификаторы наследования.

```
// ...
Vector2D operator + (const Vector2D& v);
};

class Vector3D : public Vector2D{
// ...
};

Vector3D v1, v2;
// Ошибка!
Vector3D v3 = v1 + v2;
```

Ошибка: не определено оператора сложения для класса Vector3D. Операторы наследуются. Однако в примере выше нужен оператор для класса Vector3D, однако унаследованный оператор принимает Vector2D. Ошибка станет очевиднее, если записать вызов оператора как

```
Vector3D v3 = v1.operator + (v2);
```

Закрытые (private) члены класса при наследовании всегда недоступны в производном. Члены класса с другими модификаторами доступа (protected, public) сохраняют их и в производно классе. Это правило справедливо для public-наследования.

В C++ существует ещё два вида наследования: protected и private наследование 5.5. Они соответственно уменьшают уровень доступа защищённых и открытых полей до уровней protected и private наследование. Повышение уровня доступа членов класса при наследовании невозможно.

**Перегрузка методов (overloading)** – это объявление в классе методов с одинаковыми именами при этом с различными параметрами.

```
class B {
    public:
    void bar() {cout << "bar";}
    void bar(string s) {cout << "bar s: " << s;}
    void bar(float x) {cout << "bar " << x;}
};
```

Метод bar перегружен и имеет три версии. Какая версия будет вызвана зависит от типа и количества параметров:

```
B b;

b.bar();           // bar;
b.bar(42);         // bar 42
b.bar(42 + 2);     // bar 44
b.bar("42");       // bar s: 42
b.bar("qwerty");   // bar s: qwerty
```

Перегрузка (overloading) методов

```
class B {
    public:
    void bar() {cout << "bar";}
    void bar(string s) {cout << "bar s: " << s;}
    void bar(float x) {cout << "bar " << x;}
};

class D: public B{
    public:
    void bar(int x, int y) {cout << "bar " << x << y;}

    D d;

    d.bar();           // bar;
    d.bar(42);         // bar 42
    d.bar(40, 2);      // bar 40 2
    d.bar("qwerty");   // bar s: qwerty
```

**Переопределение метода (overriding)** – объявление в производном классе метода, который заменяет собой одноименный метод базового. При этом новый метод должен иметь те же параметры что и метод базового класса.

```
class B {
    public:
    void foo() {cout << "base";}
};
```

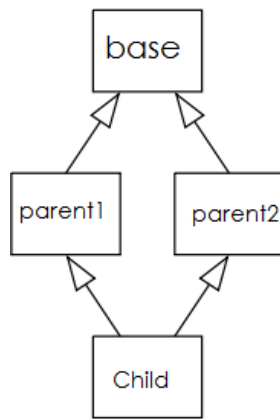


Рис. 5.6. Проблема ромбовидного наследования: если метод класса Child вызывает метод, определенный в классе A, а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C ?

```

class D: public B{
    public:
        void foo(){cout << "delivered";}
};
  
```

```

B base;
D delivered;
base.foo();           // base
delivered.foo();      // delivered
// Если нужно вызывать метод базового класса в производном:
delivered.B::foo();   // base
  
```

**Множественное наследование** – наследование от нескольких классов одновременно.

Общий синтаксис:

```

class Z: public X, public Y { . . . };
  
```

При множественном наследовании может возникать проблема неоднозначности из-за совпадающих имен в базовых классах. Поэтому лучше наследоваться от интерфейсов и классов-контейнеров.

**Deadly Diamond of Death** Проблема ромба [ wiki ]

### 5.3.1 Абстрактные классы

Если создаваемые классы не имеют общих данных, но имеют общие по смыслу и сигнатуре методы (пусть и с разной реализацией) имеет смысл создать для них базовый класс (рис 5.7), определяющий интерфейс (набор методов) – **абстрактный класс**:

```

/// Домашнее животное
class Pet{
    // это абстрактный класс, т.к. он соединит абстрактный метод
    public:
        // абстрактный метод
        virtual void say() = 0;
};

class Cat: public Pet{
    public:
        // реализация абстрактного метода
        void say() override { cout << "Мяу" << endl;};
};

class Dog: public Pet{
    public:
        // реализация абстрактного метода
        void say() override { cout << "Гав" << endl;};
};

int main(){
    // Ошибка: невозможно создать экземпляр абстрактного класса!
    Pet p;

    Cat tom;
    tom.say();        // Мяу

    Dog spike;
    spike.say();      // Гав
}

```

**Абстрактный метод** должен *определяться* со спецификатором **virtual**, а вместо тела приводится спецификатор `= 0`:

```
virtual void say() = 0;
```

Класс, у которого есть только методы не имеющие реализации (чистые виртуальные или абстрактные методы) – **абстрактный класс**. Такие классы определяют интерфейс взаимодействия, то есть набор методов (пусть не полный), поэтому подобные классы ещё называют **интерфейсами**. Экземпляры таких классов создавать нельзя ведь они не полностью определены. Наследование от этих классов, при котором для абстрактных методов определяются тела, называется реализацией.

**override** – спецификатор для переопределяемого виртуального (или абстрактного) метода в производном классе.

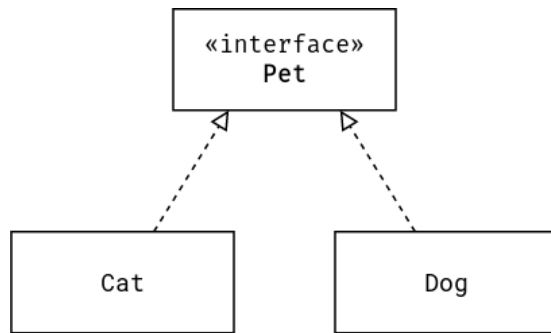


Рис. 5.7. Отношение Реализация – частный случай наследования

Не абстрактные потомки абстрактных классов обязательно должны содержать реализации всех абстрактных методов своих предков.

Ещё одна причина создавать абстрактные классы – возможность использовать классы совместно с существующим кодом, функциями и классами. Подробно об этом говорится в главе 6 SOLID, начиная с раздела 6.3 Принцип подстановки.

### 5.3.2 Когда использовать наследование

Квадрат можно назвать частным случаем прямоугольника. Однако в ООП создание класса «Квадрат» на основе класса «Прямоугольник» не оправдано: во втором (базовом) классе больше полей чем в первом (производном). Эти классы стоит построить на основе абстрактного класса «Геометрическая фигура», который задаст интерфейс своих потомков, например методы вычисления площади и периметра.

Наследование полезно для добавления новых свойств и поведения классам из существующих библиотек. Например типичный подход создания приложений с графическим пользовательским интерфейсом – создание класса, определяющего окно приложения, на основе существующего класса окна. Как правило в последнем реализовано базовое поведение: рендеринг, реакция на кнопки управления коном (свернуть, развернуть, закрыть), изменения размера и т.д.; базовые свойства: текст заголовка, цвет фона, шрифт и т. д.

**Наследование и объекты.** Наследование имеет смысл использовать, если производный класс по смыслу является более специальной версией базового, с добавлением либо полей, либо методов, либо и того и другого. Например не имеет смысла создавать на основе базового класса Собака, со свойствами: размер, возраст, окрас (см. рис 4.2) производный классы для отдельных пород собак, если индивидуальные особенности пород мо-

в стандартную библиотеку C++ не входят средства создания приложений с графическим интерфейсом. Популярны сторонние фреймворки – Qt (кроссплатформенный) и UWP для Windows [пример]. Windows Forms официально считается устаревшим с 2014 года.



гут быть описаны полями и методами одного класса. Вместо создания потомков, соответствующих породам, имеет смысл добавить поле Порода в класс Собака. В некотором смысле базовый класс и производный здесь заменяются на класс без потомков и объекты соответственно.

**Наследование и агрегация.** Один класс может включать в себя другой, но по смыслу не будет являться более частным случаем. Например класс «Событие» календаря (название, описание, дата) должен агрегировать класс «Дата», но не наследоваться от него.

## 5.4 Динамический полиморфизм

Ранее, в начале раздела 5.3 было отмечено, что при записи объекта производного класса в объект базового, первый будет преобразован к типу базового класса. Например:

```
// экземпляр базового класса
Vector2D v1(10, 20);
Vector3d v2( 3,  4, 100);

// присваивание с преобразованием объекта типа Vector2D в объект типа Vector2D
v1 = v2;
// вычисление длины вектора
v1.abs()          // -> 5.0
```

Динамический полиморфизм – это один из фундаментальных принципов ООП, который позволяет динамически выбирать реализацию вызываемого метода (из множества других методов с таким же именем и параметрами) в зависимости от подкласса объекта, для которого он вызывается. Например:

```
Vector2D *v;          // указатель на базовый класс
if (rand() % 2 == 0)
    v = new Vector3D(1,2,3) // создание экзмп. производного класса
else
    v = new Vector2D(1,2)   // создание экзмп. базового класса
// динамическое определение варианта переопределённого метода abs,
// подходящего под фактический тип объекта
v->abs();
```

Полиморфизм в широком смысле – способность функции обрабатывать данные разных типов

чётность числа, которое выдаст функция rand(), определится только на этапе выполнения программы. Следовательно тип объекта, на который указывает v, будет известен не раньше. Благодаря д. полиморфизму будет вызвана правильная реализация метода, вычисляющего длину вектора.

В отличие от статического полиморфизма на основе шаблонов или перегрузки функции, здесь фигурируют методы, а изменяющимся типом данных выступают классы. Причем классы должны иметь общего предка. Предок должен быть виртуальным или абстрактным (т.е. содержать в себе

виртуальный или абстрактный метод). При этом потомки этого класса могут содержать свои реализации виртуальных методов.

Рассмотрим пример с векторами из предыдущего раздела. Метод `abs` может быть переопределён в потомках, поэтому чтобы вызывать правильную версию метода сделаем его виртуальным в базовом классе, а в производным переопределим:

```
class Vector2D{
// ...
public:
// ...
    // объявление виртуального метода
    /// длина вектора
    virtual float abs() const {return sqrt(_x*_x + _y*_y);}
};

class Vector3D: public Vector2D{
// ...
public:
// ...
    // переопределение виртуального метода
    /// длина вектора
    float abs() const override {return sqrt(_x*_x + _y*_y);}
};
// ...
```

Соответственно класс `Vector2D` станет виртуальным.

```
int main(){
    Vector3D v2(3, 4, 10);
    Vector2D v1(100, 200);

    v1 = v2;                                // запись с преобразованием в Vector2D
    std::cout << v1.abs();                  // -> 5
}
```

Записать экземпляр `Vector3D` в экземпляр `Vector2D` без преобразования к базовому классу нельзя, в том числе потому, что экземпляр базового класса не может вместить все поля производного. Поэтому используем указатели на объекты, т.к. указатель на любой тип данных всегда имеет один и тот же размер, но может указывать на объекты любого размера:

```
Vector3D *v22 = new Vector3D(3, 4, 10);
Vector2D *v11 = new Vector2D(100, 200);

v11 = v22;
```

```
// динамическое определение типа объекта, на который указывает v1
// вызов соответствующего типу метода abs
std::cout << v11->abs();           // -> 11.18
```

Такой же механизм динамического определения варианта метода работает и для абстрактных методов (см. метод say в пример из раздела 5.3.1).

Чтобы проверить факт определения варианта метода на этапе выполнения программы запустим приведённый в начале раздела код:

```
Vector2D *v;           // указатель на базовый класс
if (rand() % 2 == 0)
    v = new Vector3D(1,2,3) // создание экзмп. производного класса
else
    v = new Vector2D(1,2)   // создание экзмп. базового класса
// динамическое определение варианта переопределённого метода abs,
// подходящего под фатический тип объекта
v->abs();
```

Реализация вызываемого метода будет зависеть не от типа указателя *v* (который *Vector2D \**), а от типа объекта на который он фактически указывает.

Каждый экземпляр виртуального класса и все его потомки содержат внутри себя дополнительное поле – указатель на таблицу виртуальных методов. Это поле создаётся компилятором автоматически. **Таблица виртуальных методов (virtual method table, VMT)** *vtable* – таблица создаваемая для каждого виртуального класса (и их потомков), которая содержит указатели на все виртуальные методы класса.

todo: Алгоритм поиска варианта виртуального метода

todo: Run-Time Type Information (RTTI) и оператор **typeid**.

Для создания полиморфных типов можно использовать в качестве базового класса не только виртуальный класс, но и его частный случай – абстрактный класс. Например:

```
/// Определяет интерфейс вектора (понятие из математики)
class Vector{
public:
    virtual float abs() const = 0;    // абстрактный метод
};

class Vector2D: public Vector{
    // ...
public:
```

```

        float abs() const override { ...реализация метода... };
};

class Vector3D: public Vector2D{
    // ...
public:
    float abs() const override { ...реализация метода... };
};

```

Экземпляры такого класса Vector создавать нельзя, но можно использовать указатели на этот класс:

```

Vector *v;
if (rand() % 2 == 0)    v = new Vector3D(1,2,3);
else                   v = new Vector2D(1,2);
v->abs();

```

## Работа с полиморфными типами

Динамический полиморфизм позволяет единообразно работать с разными подклассами. Например можно хранить в одном массиве указатели на разные классы:

```

const unsigned N = 3;
Vector* vecs[N];    // массив из указателей на объекты

vecs[0] = new Vector2D();
vecs[1] = new Vector2D();
vecs[2] = new Vector3D();

float S = 0.0;
// цикл работает корректно вне зависимости от того,
// на какие именно объекты указывают элементы массива
for (unsigned i = 0; i<N; i++)
    S += vecs[i]->abs();

S = 0.0
// аналогично можно использовать совместный цикл
for (Vector* v : vecs)
    S += v->abs();

```

Для полиморфных типов можно писать функции, зная только о базовом классе. Эти функции будут работать и со всеми производными классами (todo: примечание).

```

void foo(Vector* v){
    ...
}

```

```

}

Vector2D* v1 = new Vector2D();
Vector3D* v2 = new Vector3D();

foo( v1 );
foo( v2 );

```

См. также один из принципов SOLID: принцип подстановки Барабары Лисков.

Аналогичный подход можно использовать не только для параметров функции, но и полей класса.

Программист может создавать свои классы на основе классов из библиотек и фреймворков. Благодаря динамическому полиморфизму и наследованию (на котором он строится) новые классы могут быть успешно использованы внутри кода стороннего фреймворка.

## Динамическое преобразование типа

Благодаря таблице виртуальных методов можно вызывать только виртуальные методы или их переопределённые версии. Вызвать метод, которого нет в базовом классе нельзя:

```

Vector2D* v1 = new Vector3D();
std::cout << v1->z();    // ошибка: класс Vector2D не содержит метода z()

```

Для этого требуется операция явного преобразования типов. Воспользуемся шаблонным оператором безопасного преобразования типов **dynamic\_cast** для классов, находящихся в одной иерархии наследования:

см. также раздел 1.2.5  
Преобразования типов и  
оператор static\_cast

```

dynamic_cast<Vector3D*>( v1 ) -> z();    // -> 10

```

```

// Для удобства можно записать преобразованный указатель в переменную соответствующего типа:
Vector3D *v33 = dynamic_cast<Vector3D*>( v1 );
cout << v33 -> z();    // -> 10

```

Этот оператор, выполняет проверку возможности преобразования на этапе выполнения программы. Если преобразование невозможно, то возвращается пустой указатель. Поэтому после преобразования имеет смысл добавить проверку:

```

Vector3D *v3 = dynamic_cast<std::string*>( v );
if (v3 == nullptr)
    std::cout << "Ошибка преобразования типа";

```

Если программа гарантирует, что преобразование типов всегда будет работать корректно, то можно вызывать метод непосредственно после преобразования:

```
cout << dynamic_cast<Vector3D*>( v1 )->z();
```

Документация **dynamic\_cast**: [en.cppreference.com/w/cpp/language/dynamic\\_cast](http://en.cppreference.com/w/cpp/language/dynamic_cast)

Другие примеры: [github.com/VetrovSV/OOP/tree/master/examples/poly](https://github.com/VetrovSV/OOP/tree/master/examples/poly)

Слайды: [raw.githubusercontent.com/VetrovSV/OOP/master/OOP\\_1.2.pdf#Navigation29](https://raw.githubusercontent.com/VetrovSV/OOP/master/OOP_1.2.pdf#Navigation29)

# **6 SOLID**

**SOLID –**

[https://raw.githubusercontent.com/VetrovSV/OOP/master/OOP\\_SOLID.pdf](https://raw.githubusercontent.com/VetrovSV/OOP/master/OOP_SOLID.pdf) –  
слайды

**6.1 Принцип единственной ответственности**

**6.2 Принцип открытости и закрытости**

**6.3 Принцип подстановки**

**6.4 Принцип разделения интерфейсов**

**6.5 Принцип инверсии зависимостей**

# 7 Шаблоны проектирования

## 7.0.1 Обзор

## 7.0.2 Некоторые примеры

- синглтон (Singleton)
- Абстрактная фабрика (Abstract factory)
- Наблюдатель (Observer)
- Адаптер (Adapter)
- Декоратор (Decorator)

## Литература

Паттерны проектирования, Эрик Фримен, Элизабет Фримен

Приёмы объектно-ориентированного проектирования. Паттерны проектирования, Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес («Банда четырёх»)



## **Заключение**

# Библиографический список

1. Язык программирования C++ (с описанием C++11 и c++14). Страуструп Бьёрн. 2016 г. 1100+ стр. 2-е или более новые издания (4-е издание самое новое на момент 2021 года)
2. Эффективный и современный C++: C++11 и C++14. 42 рекомендации по использованию C++. 2016. г - 304. Скотт Мейерс
3. Буч, Г. Язык UML. Руководство пользователя / Г. Буч, Д. Рамбо, И. Якобсон; пер. с англ. Н. Мухин. – 2-е изд – Москва: ДМК Пресс, 2006. – 496 с.: ил.
4. [stepik.org/course/7/syllabus](https://stepik.org/course/7/syllabus) – stepik: Программирование на языке C++
5. Шлее М. Qt 5.10. Профессиональное программирование на C++.
6. Совершенный код (2-е издание, 2005 г.) Стив Макконнел

# Предметный указатель

access violation, 46  
ASCII, 13  
asm, 72  
assert, 60, 66  
auto, 12, 17  
cast  
    C-style, 18, 26  
catch, 56  
char, 13  
cin, 59  
class, 92  
    agregation, 117  
    association, 118  
    base, 120  
    composition, 116  
Cling, 74  
CMake, 73  
compiler-time, 5  
Conan, 74  
const, 15, 32  
copy  
    deep, 101  
cout, 19  
declaration, 9  
    forward, 92  
default, 104  
define, 49  
definition, 9  
delete, 27, 104  
delete [], 42  
diff, 74  
double, 11  
Doxygen, 35  
dynamic\_cast, 132  
enum, 16  
exception handling, 54  
float, 11, 12  
function  
    overloading, 32  
garbage collector, 6  
git, 74  
Google Test, 113  
heap, 45  
ifndef, 49  
include guards, 49  
int, 12  
memcpy, 40, 107  
memmove, 40  
memory  
    leak, 28  
MSVC, 50  
namespace, 35  
NDD, 70  
new, 27, 41  
NuGet, 74  
nullptr, 25  
override, 121  
overriding, 124  
plain old data, 107  
POD, 107  
pointer, 25  
pragma, 49  
reference, 27  
return, 29  
RTTI, 130  
run-time, 5  
segmentation fault, 46  
sizeof, 42  
SOLID, 127, 134

- sstream, 77
- stack
  - call, 46
  - memory, 45
- standard library, 76
- static, 108
- static\_cast, 18
- stod, 18
- stof, 18
- stoi, 18
- strcpy, 40
- string, 18, 40
- struct, 93
  
- template, 70
- throw, 56
- to\_string, 18
- try, 56
- type
  - safety, 18
- type checking
  - dynamic, 5
  - static, 5
- typing
  - weak, 6
  
- undefine behavior, 9
- undefined behavior, 39, 40
- Unicode, 14
- unit test, 113
  - Google Test, 113
  
- Visual Studio, 113
- vtable, 108
  
- Комплексные числа, 12
- абстрагирование, 90
- абстрактный тип данных, 88
- автоматическое тестирование, 60
- ассемблер, 72
- библиотека
  - стандартная, 76
- геттер, 95
- декремент, 13
- директивы препроцессора, 49
- единица трансляции, 34, 49
- запись
  - экспоненциальная, 12
- инициализация, 9

- списки, 104
- инкапсуляция, 94
- инкремент, 13
- исключительная ситуация, 54
- класс, 92
  - абстрактный, 125, 130
  - базовый, 108, 120
  - виртуальный, 129
  - дружественный, 110
  - локальный, 97
  - отношение агрегации, 117
  - отношение ассоциация, 118
  - отношение композиция, 116
  - отношения реализация, 126
  - подкласс, 120
  - производный, 120
  - шаблонный, 78
  - экземпляр, 95, 104
- компоновщик, 51
- константа
  - безымянная, 11
  - математическая, 16
- конструктор, 98
- копирование
  - глубокое, 101
  - поверхностное, 101
- куча, 41, 45
- линкер, 51
- литерал, 11
  - целый, 12
  - целый восьмеричный, 12
  - целый двоичный, 12
  - целый шестнадцатеричный, 12
- макрос, 50
- мантисса, 12
- массив, 38
  - динамический, 41
  - статический, 42
- метод, 92
  - абстрактный, 126
  - виртуальный, 129, 130
  - переопределение, 129
- методы
  - виртуальные, 108
  - таблица виртуальных, 108
- модуль
  - cstring, 40
- наследование, 108

- многоуровневое, 122
- неопределённое поведение, 9, 39
- объект, 95
  - идентичность, 105
  - константа, 98
  - массив, 105
  - равенство, 104
- объявление, 9
  - предварительное, 92
- операнд, 21
- оператор, 21
  - перегрузка, 122
  - тернарный условный, 22
- операторы
  - логические, 22
  - присваивания, 21
- определение, 9
- память, 108
  - утечка, 27, 28, 46
- параметр-константа, 32
- переопределение, 121, 124
- перечисление, 16
- поведение
  - неопределённое, 25
- поле, 92
- полиморфизм
  - динамический, 128
  - статический, 70
- поток
  - строковый, 77
- предварительное объявление, 118
- предусловия, 56
- препроцессор, 51
- принцип
  - сокрытия, 94
- принцип
  - единственной
    - ответственности, 30, 56
  - сокрытия, 95
- программирование
  - обобщённое, 70
- профилирование, 87
- разыменование, 25, 39
- сборщик мусора, 6
- сеттер, 95
- символ
  - экранирование, 15

- система сборки, 73
- ссылка, 27
- стек
  - вызовов, 46
  - область памяти, 45
- строка, 18, 40
  - поток, 77
- структура данных
  - простая, 107
- тест
  - автоматический модульный, 113
- тестирование
  - автоматическое, 65
  - модульное, 69
- тип, 11
  - подтип, 120
  - преобразование, 18
  - символьный, 13
  - строковый, 14
- типизация
  - слабая, 6
  - динамическая, 5
  - сильная, 6
  - статическая, 5
- типобезопасность, 18
- указатель, 25, 42
  - void \*, 40
  - висячий, 27
  - на void, 26
  - умный, 78
- утечка памяти, 102
- файлы
  - бинарные, 84
  - заголовочные, 96
  - объектные, 51
- функция
  - дружественная, 110
  - перегрузка, 32, 72
  - шаблонная, 70
- числа
  - магические, 15
- число
  - магическое, 39