версия от 11 ноября 2022 г.

С. В. Ветров

Черновик учебного пособия

Объектно-ориентированное программирование на C++

Оглавление

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

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 поход 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 типизация 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, Hackell, Pyhton. 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 (Pyhton) 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 – краткое описание стандартов

1.1.1 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

1.1.2 Компиляторы

- G++ (MinGW-64)
- MSVC
- Clang
- Intel C++ Compiler
- ...

1.2 Основы

1.2.1 Алфавит

1.2.2 Структура программы

Минимальный вариант главной функции программы:

о параметрах командной строки и возвращаемом значении функции main см. раздел 1.10

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

    // ...основной алгоритм...

    return 0;
}
```

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 приводится до объявления функции main.

Типичная структура программы:

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

```
/// 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...
}
```

1.2.3 Типы данных

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 фунции или метода.

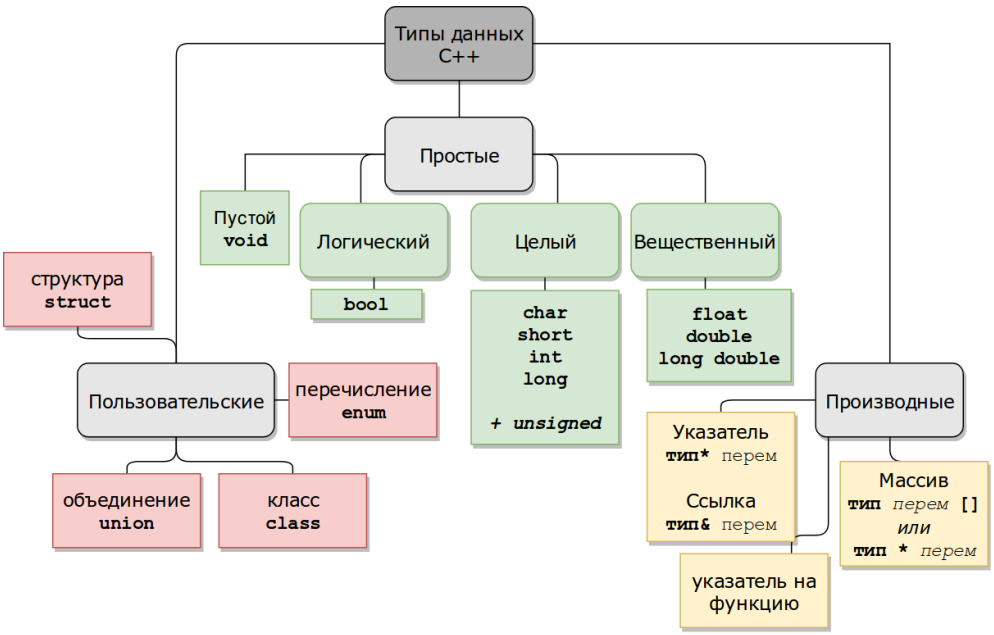


Рис. 1.1. Типы данных

В C++, в отличии от Паскаля, нет специального раздела программы для определения или объявления переменных. Синтаксис объявления переменной[1]:

```
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;
```

Подробнее о простых типах: ru.cppreference.com/w/cpp/language/types

1.2.4 Числовые типы данных

Размер памяти, занимаемой, типами long, double, long double и другими многобайтовыми типами данных может отличаться в зависимости

[1] en.cppreference.com/w/cpp/language/declarations

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]

от разрядности (32 ил 64 бита) платформы, для который компилируется программа. Размер значения типа `bool` – 1 байт, хотя, фактически, для такой переменной достаточно одного бита. Но ОС может адресовать память только по байтам, но не по битам.

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
```

1.2.5 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:

Больше информации о целочисленных литералах, в том числе суффиксах: 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.

```
// 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;
```

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

Комплексные числа todo:

1.2.6 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_incdec

…

1.2.7 Символьный тип и строковые литералы

char – символьный тип, занимает один байт. В 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.

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++:

напомним, что для кодирования любой шестнадцатеричной цифры необходимо 4 бита

```
// 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");
```

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

см. также раздел 1.8.2 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
```

1.2.8 Константы и выражения времени компиляции

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

см. также раздел 1.5.4 Спецификатор const и указатели

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

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 из 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, конвенциальное имя для размера массива

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

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

Математические константы в C++20 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 [2] о константах.

1.2.9 Перечисления (enum)

Плохая практика – вводить не именованные условные обозначения

```
/// 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++[2] есть специальный тип данных, который помогает вводить наборы однотипных обозначений – *перечисления* (enum).

[2] 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 [2].

todo: enum class, or enum struct

Документация: en.cppreference.com/w/cpp/language/enum

1.2.10 auto

Указание **auto** вместо типа заставляет компилятор самостоятельно подставить тип ориентируясь на задаваемое значение.

определение типа 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!
```

1.2.11 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

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;

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: ...

1.2.12 Приоритет операторов

todo: ...

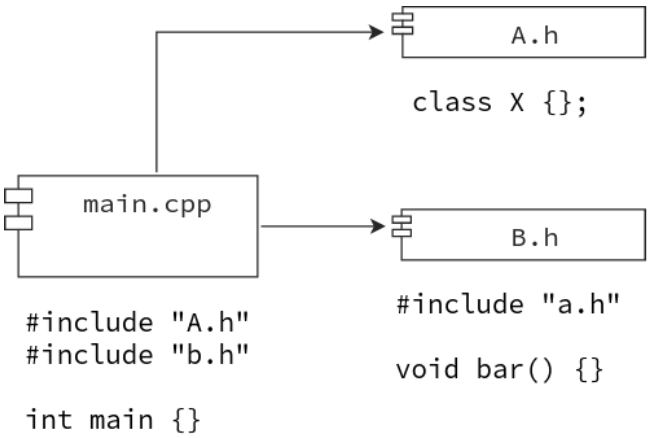
1.2.13 Подключение заголовочных файлов

include

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

- Типы данных объявлять в заголовочных
- Переменные объявлять с extern
- using namespace и include *только* при необходимости
- ...

Защита от повторного подключения



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 подключения 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

class X{};
// ...

#endif
```

1.2.14 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 вывода. 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)

# 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);
```

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/iomanip/setprecision/

Про обработку ошибок при вводе значений идет речь в параграфе **?? ??**.

1.2.15 Операторы

Оператор –

Операнд –

1.2.16 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
```

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
```

Документация: en.cppreference.com/w/cpp/language/operator_other

1.2.17 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 << " ";
```

1.3 Компиляция программы

1.3.1 Компиляция программы из одного файла

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

```cpp
#inlude <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 появится 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 праметр `std`:

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

1.3.2 Макросы препроцессора

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: https://learn.microsoft.com/ru-ru/cpp/build/reference/zc-cplusplus?view=msvc-170

1.3.3 Этапы компиляции

1. **Препроцессинг** . Обработки директив *препроцессора* C++: `include` `define`, `ifdef`, и др. На этом этапе, в том числе, происходит вставка содержимого файлов указанных в директивах `include`.
2. Преобразование в Ассемблерный код.
3. Преобразование в машинный код. В результате создаются *объектные файлы* из всех cpp фалов 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

1.3.4 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_uni1.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 имена указаны в директивах `include`.

В программе, компилируемой GCC (MinGW), можно вывести версию последнего самого нового поддерживаемого стандарта [en.cppreference.com/w/cpp/preprocessor/replace]

```
std::cout << __cplusplus << std::endl;          // 201703  // (C++17)
```

В MSVC `__cplusplus = 119711` вне зависимости от поддерживаемого стандарта [docs.microsoft.com/en-us/cpp/build/reference/zc-cplusplus?view=msvc-160]

1.4 Модули C++20

единица трансляции –

1.5 Динамическая память, указатели и ссылки

1.5.1 Указатели

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

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

Другими словами: указатель – переменная которая хранит адрес памяти, в том числе может хранить адрес памяти, где находится другая переменная.

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;
```

Основные операции для работы с указателями:

- Взятие адреса. Оператор &; используется при записи адреса переменной в указатель.
- Разыменование. Оператор *
  – обращение к значению, адрес которого записан в указателе

```
// 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 другие числовые значения для указателя недопустимы
ip = 1235489;

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 обратится 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

здесь используется преобразование типов C-style cast; для преобразования типов одинакового размера (в том числе в массивы) см. union

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

```
Результат: 13 12 11 10
```

В C++ рекомендуется избегать использование таких указателей, если есть альтернатива.

1.5.2 new и delete

todo:

malloc vs new

1.5.3 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 3.2 «Умные указатели» о типах данных, автоматически очищающих выделенную память.

1.5.4 Спецификатор const и указатели

Указатель на константу:

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

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

      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

1.5.5 Проблемы динамической памяти

перезаписывание указателя

потеря указателя

Утечка памяти (memory leak)

неосвобождённая память

см. также раздел про умные указатели

1.6 Функции

Общий вид определения (definition) функции.

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

В некоторых компиляторах возраст значения обязателен, если тип возвращаемого значения не пустой (void)

Если возвращаемый тип функции **void**, то после **return** не должно быть никакого выражения.

Формальные параметры –

Фактические параметры –

Возврат значения из функции

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

// вызов функции
int y = foo( 100 );


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

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 1.8.4 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: 1.8.4 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 значению 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 отличатся 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

1.6.1 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 отличатся 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.

1.6.2 Значения параметров по умолчанию

1.6.3 inline-функции

...

1.6.4 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.

1.6.5 Спецификатор constexpr

`constexpr` –

1.6.6 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 стремится 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 ivalid_argument если h==0
float bmi(float m, float h);

// отлично (машинно-читаемый комментарий для
// системы документирования Doxygen):
/// вычисляет индекс массы тела;
/// бросает исключение ivalid_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 [2].

См. также параграф **?? ??**.

1.7 Пространства имён (namepaces)

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.

1.8 Массивы

1.8.1 Статические массивы

```
// объявление массива
int a[128];
int b[256];

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

Тип таких массивов описывается как `Type[N]`, т.е. содержит количество элементов. Два статических массива с одинаковым типом элементов но разным их количеством (`a` 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 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]`) не обязательно в стандарте регламентируется как *неопределённое поведение* (undefined behavior, UB). Программа может аварийно завершится или продолжить выполнение с непредсказуемыми последствиями. Например, если в оперативной памяти 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 считаемся задавать размер массива не через *магическое число* (magical number), а с помощью константы. Это упрощает модификацию и пониманию 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
```

1.8.2 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 посимвольною:

```
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`.

1.8.3 Функции для работы с массивами

Функции копирования (`strcpy`) и др. функции преобразования строковых массивов: 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 их 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
```

1.8.4 Динамические массивы

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;

int *z = new int;           // выделение памяти под одно значение типа int

// 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

// 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[] a` 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 отличии от одиночных значений, хранящихся в куче, освобождать память занимаемую массивом нужно оператором `delete[] a;`. Вызов оператора `delete` для массива не считается синтаксической ошибкой, но освободит 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 изменятся, 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 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);

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

    return 0;
}
```

1.9 Устройство памяти программы

- Статическая память (data на рис.)
- Динамическая память (heap, куча)
- Автоматическая паять (stack, стек)
- Сегмент кода (text на рис.)

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

access violation, **segmentation fault** –

Стек вызовов (call stack) – …

1.10 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[])
```

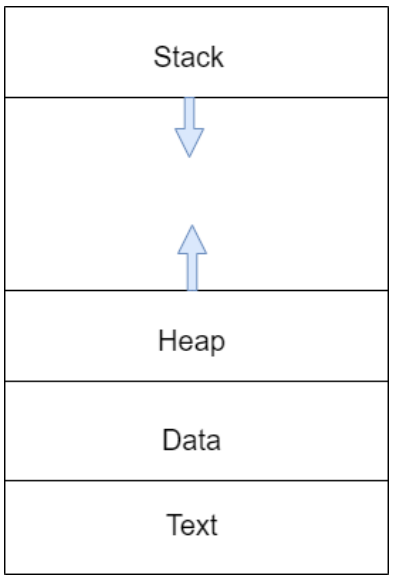


Рис. 1.4. Устройство памяти программы

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

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();
const char *ss = s.c_str();
string s3 = s1 + s2;
s1 += s2;
```

3.2 Умные указатели

Эти шаблонный класс, который автоматически освобождает память объекта, которым владеет.

```
#include <memory>

std::unique_ptr
std::make_unique
```

3.3 Контейнеры

3.3.1 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 переменой 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
```

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

```
    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;
}
```

3.3.2 Сравнение

3.4 Файловые потоки

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


// класс ifstream -- для чтения файлов (input filestream)
ifstream in;

// класс ofstream -- для записи в файлы (output filestream)
ofstream out;

// класс fstream -- для чтения и записи
in_out;
```

Запись в файл

```
#include <fstream>
using namespace std;
...
// создать объект для записи в файл
// и открыть текстовый файл для записи
ofstream f("myfile");
// запись в файл
// здесь все данные будут записаны слитно. так лучше не делать
f << "qwerty";
f << 123;
f << 3.14;
f << endl; // записать символ перехода на новую строку
f << 42.5;
f.close();
```

Содержимое созданного файла:

```
qwerty1233.14
42.5
```

https://en.cppreference.com/w/cpp/io/basic_ofstream

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

Построчное чтение файла

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

// ...
ifstream f;
f.open(filename);
if (f.is_open()){
    string buf;
    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: https://github.com/VetrovSV/OOP/blob/master/2021-fall/bin-files.md

4 Введение в объектно-ориентированное программирование

5 Отношения между классами

5.1 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;}
};
```

Агрегация – более слабый вариант композиции, когда время жизни части и целого не связаны. Можно создать объект Whole не создавая объекта Part.

Рис. 5.1. Отношение Композиция

```
class Part{
    // ...
};

class Whole{
    Part *filed1;
}
```

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 filed1[10];
}
```

Композиция:

```
class Part{
    // ...
};

class Whole2{
    Part *filed1;            // указатель на массив
    // или
    vector< Part > filed2;
    // или
    vector< Part* > filed3;  // массив из указателей
    // ...
}
```

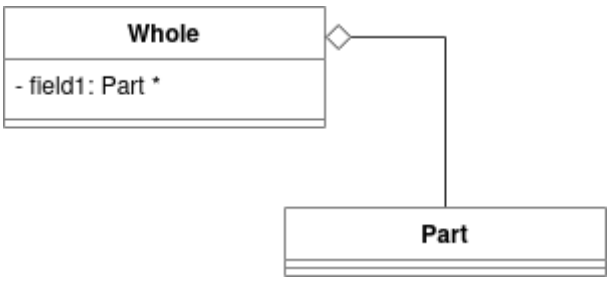


Рис. 5.2. Отношение Агрегация

5.2 Наследование

Пример. Один класс. Опишем класс для вектора $V = (v_x, v_y)$ на плоскости

члены класса с модификатором доступа protected доступны только классу и его потомкам (производным классам)

```
/// 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);}
};
```

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 5.3.

При наследовании в новый класс переходят все поля базового класса и все его не специальные методы вне зависимости от модификатора доступа.

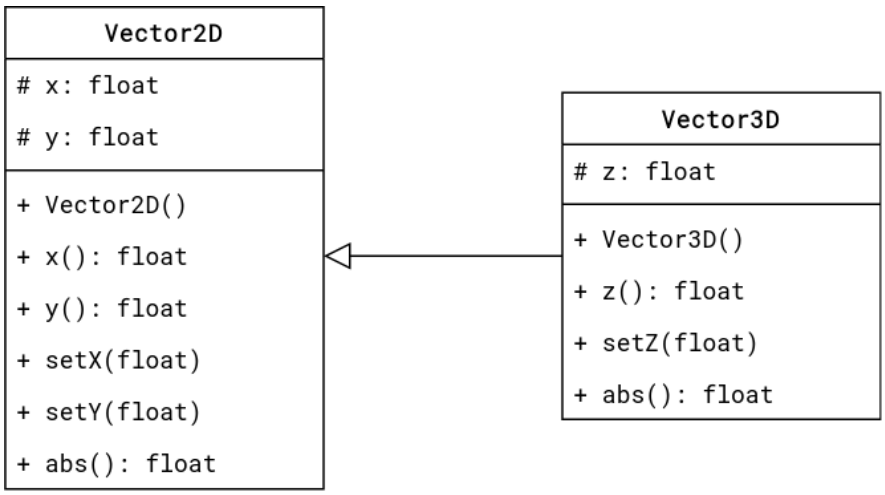


Рис. 5.3. Наследование на UML диаграмме классов. Для Vector3D (производный класс) приведены только его собственные поля, без полей унаследованных от Vector2D (базовый класс)

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 унаследованого 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
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 вызовов 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:
    // ...
    Vector2D operator + (const Vector2D& v);
};

class Vector3D : public Vector2D{
    // ...
};

Vector3D v1, v2;
// Ошибка!
Vector3D v3 = v1 + v2;
```

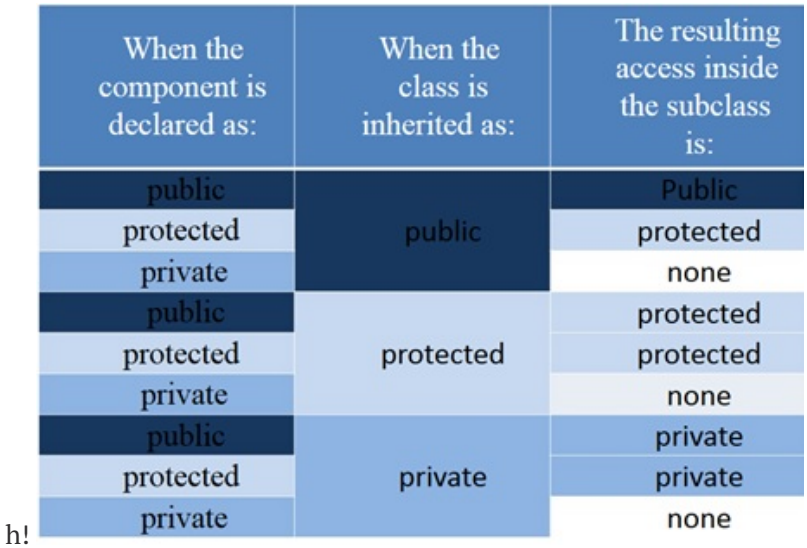
Ошибка: не определено оператора сложения для класса Vector3D. Операторы наследуются. Однако в примере выше нужен оператор для класса Vector3D, однако унаследованный оператор принимает Vector2D. Ошибка станет очевиднее, если записать вызов оператора как

```
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 5.4. 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.



h!

Рис. 5.4. Модификаторы наследования.

```
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";}
};

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]

5.2.1 Абстрактные классы

Если создаваемые классы не имеют общих данных, но имеют общие по смыслы и сигнатуре методы (пусть и с разной реализацией) имеет смысл

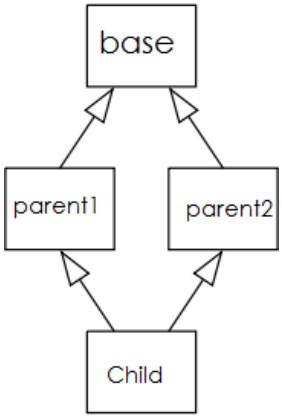


Рис. 5.5. Проблема ромбовидного наследования: если метод класса Child вызывает метод, определенный в классе A, а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C ?

создать для них базовый класс (рис 5.6), определяющий интерфейс (набор методов) – **абстрактный класс**:

```
/// Домашнее животное
class Pet{
    // это абстрактный класс, т.к. он соедржит 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
}
```

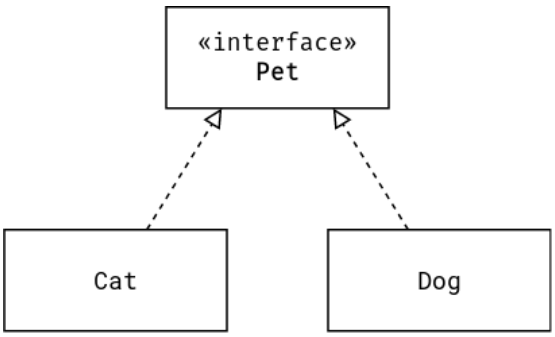


Рис. 5.6. Отношение Реализация – частный случай наследования

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` –

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.

5.2.2 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, но 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.

в стандартную библиотеку C++ не входят средства создания приложений с графическим интерфейсом. Популярны сторонние фреимворки – Qt (кросс-платформенный) и UWP для Windows [пример]. Windows Forms официально считается устаревшим с 2014 года.

Наследование и объекты. Наследование имеет смысл использовать, если производный класс по смыслу является более специальной версией базового, с добавление 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 **??**) производный классы для отдельных пород собак, если индивидуальные особенности пород могут 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 является более частным случаем. Например класс «Событие» календаря (название, описание, дата) должен агрегировать класс «Дата», но не наследоваться от него.

5.3 Динамический полиморфизм

github.com/VetrovSV/OOP/tree/master/examples/poly

6 SOLID

SOLID –

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

Заключение

Библиографический список

1. 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.
2. Совершенный код (2-е издание, 2005 г.) Стив Макконнел
3. stepik.org/course/7/syllabus – stepik: Программирование на языке C++

Предметный указатель