версия от 16 октября 2022 г.

С. В. Ветров

Черновик учебного пособия

Объектно-ориентированное программирование на C++

Оглавление

Введение

Пособие освещает только основы языка C++.

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
```

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++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.

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.

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

    // ...основной алгоритм...

    return 0;
}
```

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

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

В дальнейших примерах, для краткости, функция объявление функции 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.

1.2.3 Типы данных

Объявление (declaration) –

Определение (definition) –

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

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

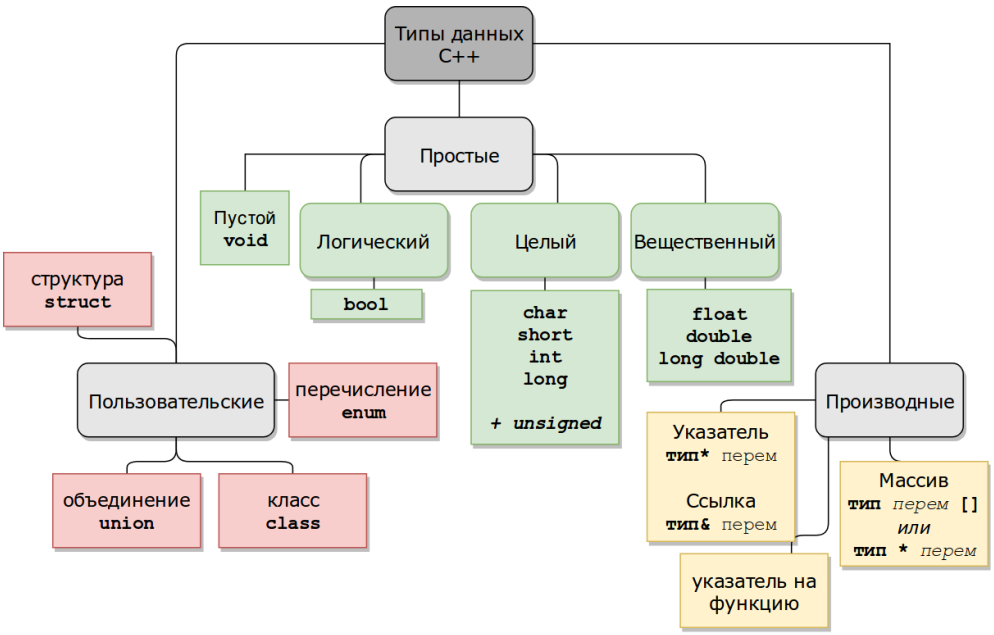


Рис. 1.1. Типы данных

- init-declarator-list – набор идентификаторов с необязательной инициализацией

Инициализация –

Примеры объявления переменных и констант:

```
int n;
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 от разрядности (32 ил 64 бита) платформы, для который компилируется программа.

en.cppreference.com/w/cpp/language/sizeof

```
int n;              // можно не задавать значение
float x = -47.039;  // можно задавать...
float y,z;
const short N = 24;  // константе задавать значение обязательно
char str1[] = "qwerty"; // строка как массив символов
// string - тоже строка, но лучше (удобнее в работе)
string str2 = "qwerty"; // нужно подключить модуль string
n = N;
```

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. Числовые типы данных [компилятор ??? для 64-разрядной ОС]

```
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.

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, приведено в документации: en.cppreference.com/w/cpp/language/integer_literal.

Шестнадцатеричная запись числа удобна тем, что 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

1.2.6 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;
```

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.

1.2.7 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).

```
// 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

[2] как и во многих других языках программирования

```
// 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

1.2.8 auto

Указание **auto** вместо типа заставляет компилятор[3] самостоятельно подставить тип ориентируясь на задаваемое значение.

```
auto x = 20;            // int
auto y = 3.14159;       // float
auto z = "gues type";   // char*
auto a; // ошибка! Не задано значение!
```

1.2.9 Преобразования типов

```
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

[3] определение типа статическое, то есть до запуска программы!

Типобезопасность(type safety) –

C-style cast – ...

```
static_cast < new-type > ( expression )
```

`static_cast` безопасно (с проверкой соответствия типа) преобразовывает выражение одного типа в другой:

```
static_cast < new-type > ( expression )
```

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: ...

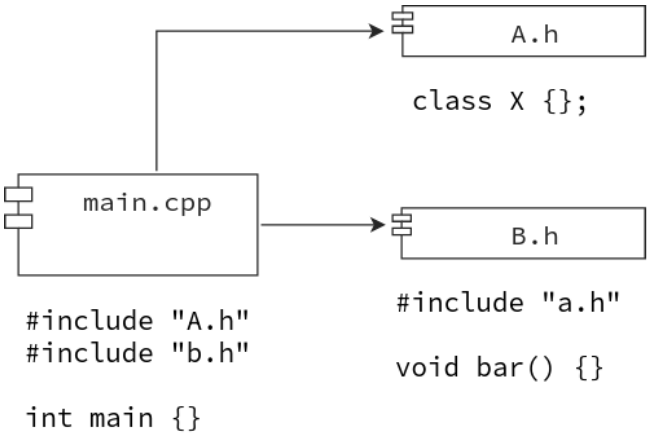
1.2.10 Подключение заголовочных файлов

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
```

1.2.11 Ввод и вывод на экран

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.

Настройка вывода

```
#include <iostream>     // std::cout, std::fixed
#include <iomanip>      // std::setprecision

...
// Установка формата вывода:
// (без использования экспоненциальной формы)
// установка 2 знаков после запятой
cout << fixed << setprecision(2);


// Вывод строки и перменной одновременно
cout <<  "X = " << x << endl;
```

Вывод

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

`<<` – оператор чтения данных с клавиатуры.
Левый операнд – объект cin;
Правый операнд – переменная.

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);
```

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 2.1
Пример обработки исключений при консольном вводе.

1.2.12 Операторы

1.2.13 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++.

```
#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 Процесс 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

1.3.3 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; также 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

```
// объявление указателя на тип int
int * ip;

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
// чтобы обратится не к адресу, который записан в указателе
// а к значению, на которое он указывает нужно использовать
// оператор разыменования *
*ip = 8; // переменная i теперь содержит 8

int *ip2;

// конечно 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
```

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 информация 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++ использовать такие указатели не рекомендуется.

Указатели и модификатор const

```
int a, b;
const int c = 42;

int *p1 = &a;
p1 = &c;             // ошибка: не может указывать на константу

// указатель с запретом изменять значение по своему адресу
const int *p2;
p2 = &a;             //         можно изменять сам указатель
*p2 = 43;            // ошибка: нельзя изменять значение по адресу указателя

p2 = &c;             //         может указывать на константу

// постоянный указатель int
int *const p3 = &a;
p3 = &b;             // ошибка: нельзя изменять адрес
*p3 = 43;            //         можно изменять значение по адресу
```

```
// указатель-константа значение по адресу которого нельзя изменять
const int *const p4 = &a;
```

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

1.5.4 ***

todo: константы + ссылки указатели

1.5.5 Проблемы динамической памяти

Утечка памяти (memory leak)

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 1.8.2 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: 1.8.2 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 inline-функции

...

1.6.2 Спецификатор constexpr

C++constexpr –

1.6.3 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 Динамические массивы

Динамические массивы в C++ – это переменные указатели, а память под них выделяется оператором **new** во время выполнения программы в *куче*.

Пример.

```
unsigned n = 128;

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;
```

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 `delete[] a;` 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);
```

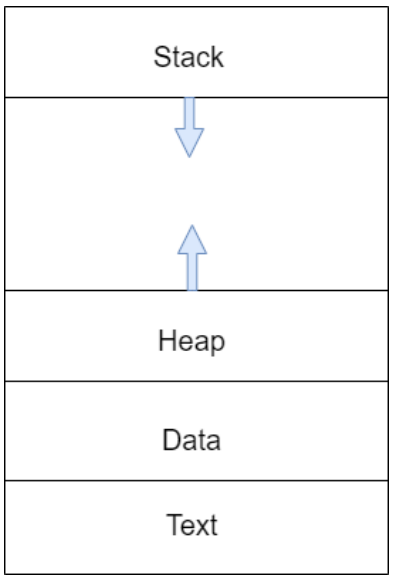


Рис. 1.3. Устройство памяти программы

```
    // 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

Стек вызовов (call stack) – ...

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 либо предусмотреть другой способ сообщения об исключительной ситуации внутри функции - например через параметр. Если исключительная ситуация и возможность её обработки возникают на разном уровне вложенности вызова функций:

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

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`**, то программа бы завершилась аварийно:

```
terminate called after throwing an instance of 'int'
```

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] из стандартной библиотеки, описанный в заголовочном файле (`stdexcept`).

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 {
        // защищуный 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: `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.

Функция `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();
}
```

Алгоритм выбора подходящего обработчика catch

...

Вызов throw внутри обработчика исключительных ситуаций throw

...

Производительность и throw

...

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 находится на разных уровнях вложенности вызова функций. Т.е. исключительная ситуация может возникнуть внутри 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.

Оператор `assert` используется для автоматических тестов, 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$_{a}rguments$
- Описание и примеры:
  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 отельную 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 это 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) mutable исключения атрибуты -> 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

...

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:
$x = (1, 2, 3, 4, 5, 6)$, $std = 1.707825127659933$
$x = (216.68167942, -472.1980655, -155.84417437, 641.71416684, -330.48466343, 426.13143811, -565.02650103, 730.202725, -655.47498993, -336.26766006)$,
$std = 486.51128339480533$
$x = (42)$, $std = 0$
$x = (100, 200)$, $std = 50$

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 ε. 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 условие 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 выполнятся 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";
}
```

todo: выключение assert

todo: assert с выводом сообщения (и оператор ,)

todo: TDD

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

Ссылки

1. О тестировании: 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

2.4 Умные указатели

2.5 Ассемблерные вставки

Общий синтаксис вставки ассемблерного кода для компилятора MSVC:

```
__asm {
    asm-code
};
```

Пример:

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

```
#include <iostream>

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

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

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

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 Контейнеры

3.2.1 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
    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 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

4.1 Предпосылки появления ООП

...

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 АТД 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

4.3 Классы в C++

Метод –

Поле –

Общий синтаксис объявления класса:

объявление класса в документации языка: en.cppreference.com/ /w/cpp/language/class

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

        member-specification
   };
```

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 (для наследования).
- **`private`**, **`protected`**, **`private`** – области видимости: открытая (доступна всем), защищённая (доступна классу и потомкам), private (доступна классу).
- `member-specification` – описание членов класса

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

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 (приниципп 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 отличии 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.

Пример работы с экземплярами класса

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

// 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:

```
#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 гарантирвать, 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[1].
- 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).
- 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 преобразования (Conversion constructor)

Общий вид:

```
ClassName(T t)
```

T – некоторый тип

[1] конструктор в python называется `__init__`

- Принимает один параметр
- Тип параметра должен отличатся 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

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 заботится об освобождении памяти массива. Перед завершением функции process она очистит память, занимаемую всеми локальными переменными. Перед удалением локального объекта a 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 ссылать 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

Списки инициализации

спецификаторы default и delete

4.3.2 Статические члены класса (static members)

Статические члены класса доступны без объявления объекта.

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

документация: en.cppreference.com/w/cpp//la

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

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

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.

4.3.3 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;
};
```

4.3.4 Перегрузка операторов

О перегрузке оператором см. также раздел 2.2.4 Перегрузка операторов (operator overloading).

Перегрузка оператора сложения для класса Seconds[2]. Оператор сложения может быть определён как[3]:

Он принимает два аргумента (операнда), типы которых могут отличаться и возвращает новое значение, тип которого совпадает с первым операндом.

seconds.h

```
/// Класс для отсчёта секунд на отрезке от 0 до 59 с переполнением
class Seconds{
```

[2] полный пример: github.com/VetrovSV/OOP/tree/master/examples/simple$_{c}lass$
[3] en.cppreference.com/w/cpp/language/operator$_{a}rithmetic$

```
    /// 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:

```
sc = sa + sb;
cout << sc.toString();      // 8
```

Оператор сложения как член класса **seconds.cpp**

```
class Seconds{
    /// количество секунд
    short s;

    // ...

Seconds Seconds::operator +(const Seconds& s2) const;
}
```

...

4.3.5 представление класса в UML

Порядок вызова конструкторов и деструкторов

4.4 Шаблонные классы

4.5 Наследование

4.6 Отношения между классами

Диаграмма классов в UML

4.7 Динамический полиморфизм

5 SOLID

SOLID

6 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

Предметный указатель