версия от 25 сентября 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 актуальны для стандарта C++19.

# 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)
- Большое сообщество программистов, большая коллекция библиотек, справочной информации
- Популярен в течении последних 30+ лет, развитая стандартная библиотека
- Активно развивается: новые стандарты выходят почти каждые 2 года. C++03, C++11, C++14, C++17, C++11, C++20

- Множество реализаций для всех популярных ОС. компиляторы: MSVC, GCC, Clang\LLVM, Intel C++ compiler, ...
- 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
- ...

### 1.1.3 Алфавит

### 1.1.4 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.

### 1.1.5 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
- init-declarator-list – набор идентификаторов с необязательной инициализацией

***Инициализация*** –

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

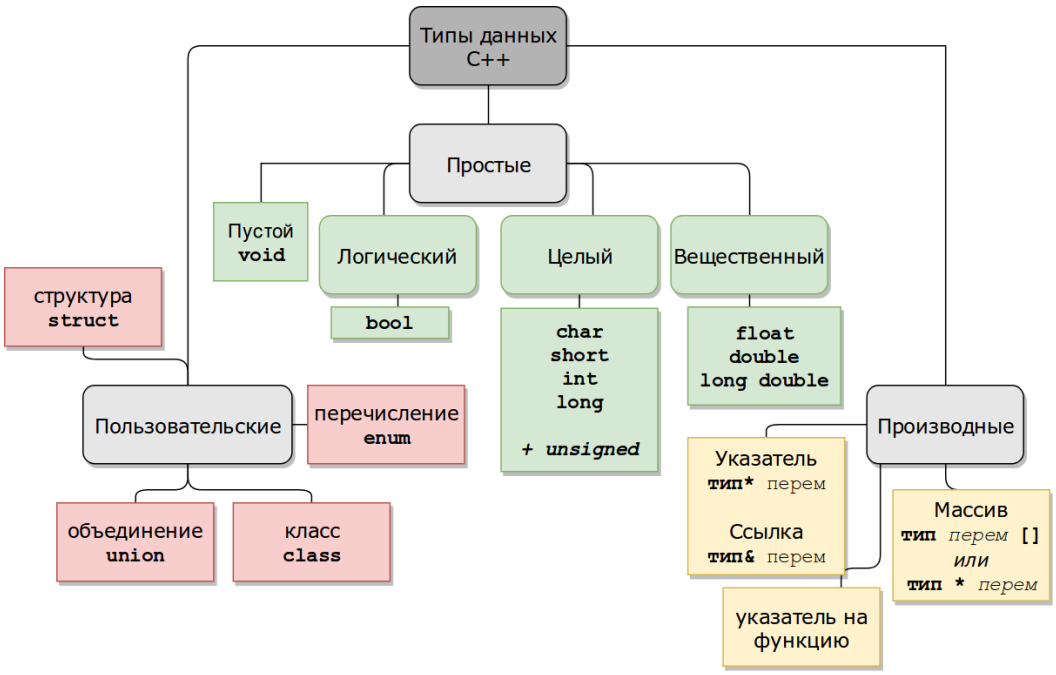


Рис. 1.1. Типы данных

Примеры объявления переменных и констант:

```
int n;
const float x = 1.68;
```

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

### 1.1.6 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;
N = n;  // Ошибка! Константу поменять нельзя
a = 42; // Ошибка! Переменная a не объявлена
```

| 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

### 1.1.7 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 суф-фиксах, приведено в документации: en.cppreference.com/w/cpp/language/integer_

**Вещественные литералы** обязательно всегда 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;
```

### 1.1.8 Константы

Математические константы в C++20 В стандарте C++20 введена библиотека математических констант numbers: https://en.cppreference.com/w/cpp/nu

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

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

### 1.1.9 auto

Указание **auto** вместо типа заставляет компилятор[2] 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!
```

### 1.1.10 Функции преобразования типов

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

см также stoi, stod и т.п. cplusplus.com/reference/string/string/

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

### 1.1.11 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 за
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 Компиляция программы

### 1.2.1 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
```

### 1.2.2 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 свя-

зи между данными среди множества других объектных файлов и создать единый исполняемый файл из них.

Детальное описание процесса компиляции: en.cppreference.com/w/cpp/language/t

### 1.2.3 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/cp

```
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]

## 1.3 Динамическая память, указатели и ссылки

### 1.3.1 Указатели

**Указатель** (pointer) – переменная, диапазон значений которой состоит из адресов ячеек памяти или специального значения – нулевого адреса (`nullptr`)[3].

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 *
```

[3] до C++11 вместо `nullptr` использовался идентификатор `NULL`, который был определён директивой препроцессора: `##define NULL 0`

```
*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
```

### 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;
```

## 1.3.2 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!
```

## 1.4 Массивы

### 1.4.1 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
```

### 1.4.2 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 и 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.

**Передача массивов в функции**. Статические массивы в функции передавать проблематично, из-за того что их тип содержит инфор-

мацию 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!
}
```

### 1.4.3 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
    // двумерный массив -- это массив из массивов
    int * *matr = new int*[N];
    // выделение памяти под двумерные массивы (строки матрицы)
    for (int i = 0; i < N; ++i)
        matr[i] = new int[M];
```

```
    print_matr(matr, N, M);

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

    return 0;
}
```

# 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:

```
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);
    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.

Обработка этих исключительных ситуаций опишем в той части программы, где происходит взаимодействие с пользователем:

```
#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/ 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();
}
```

# 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
    // 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

### Файлы

```
#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 файл
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

# 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.

# Предметный указатель